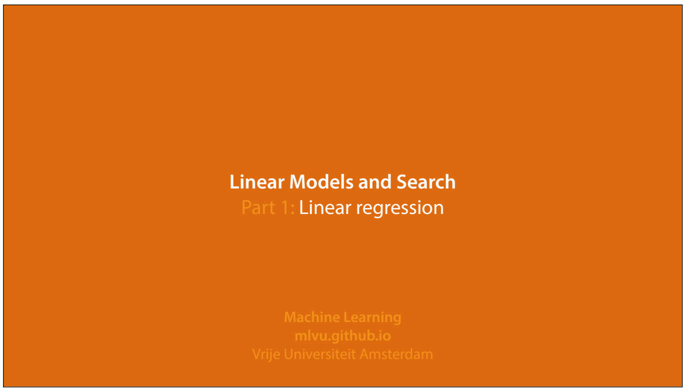
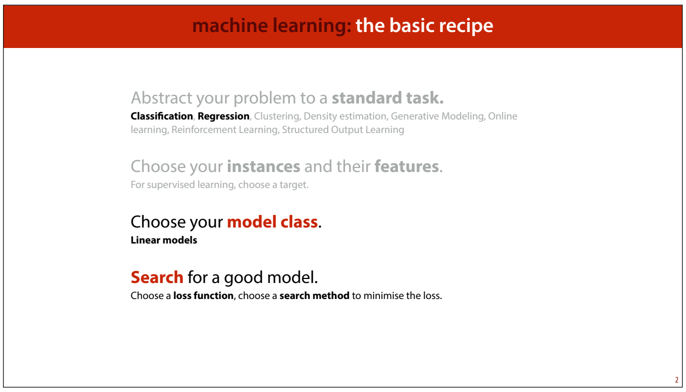


Linear Models and Search

Part 1: Linear regression

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

|section|Linear regression|
|video|https://surfdrive.surf.nl/files/index.php/s/WFvDwltABNZVm60/download|



machine learning: the basic recipe

Abstract your problem to a **standard task.**
Classification, **Regression**, Clustering, Density estimation, Generative Modeling, Online learning, Reinforcement Learning, Structured Output Learning

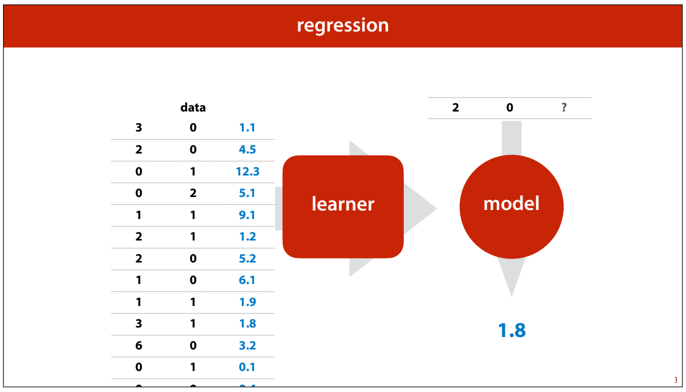
Choose your **instances** and their **features**.
For supervised learning, choose a target.

Choose your **model class**.
Linear models

Search for a good model.
Choose a **loss function**, choose a **search method** to minimise the loss.

Here is the "basic recipe" for machine learning we saw in the last lecture. In this lecture, we’ll have a look at **linear models**, both for regression and classification. We’ll see how to define a linear model, how to formulate a loss function and how to search for a model that minimizes that loss function.

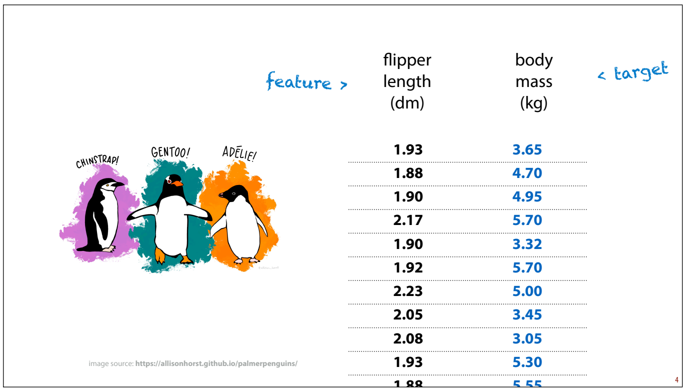
Most of the lecture will be focused on *search methods*. The linear models themselves aren’t that strong, but because they’re pretty simple, we can use them to explain various search methods that we can also apply to more complex models as the course progresses. Specifically, the method of **gradient descent**, which we'll introduce here, will be the search method used for almost all approaches we will discuss.



regression

	data	
3	0	1.1
2	0	4.5
0	1	12.3
0	2	5.1
1	1	9.1
2	1	1.2
2	0	5.2
1	0	6.1
1	1	1.9
3	1	1.8
6	0	3.2
0	1	0.1

We’ll start with **regression**. Here’s how we explained regression in the last lecture.



feature >

< target

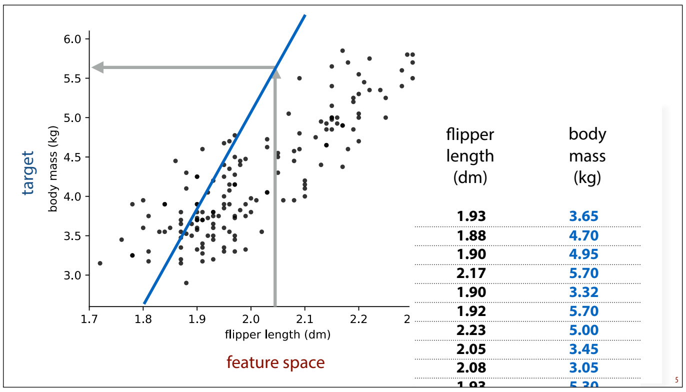
flipper length (dm)	body mass (kg)
1.93	3.65
1.88	4.70
1.90	4.95
2.17	5.70
1.90	3.32
1.92	5.70
2.23	5.00
2.05	3.45
2.08	3.05
1.93	5.30
1.88	5.55

image source: https://allisonhorst.github.io/palmerpenguins/

This is is the example data we used to illustrate regression: predicting the body mass of a penguin from its flipper length.

data source: **https://allisonhorst.github.io/palmerpenguins/**, **https://github.com/mcnakhaee/palmerpenguins** (python package)

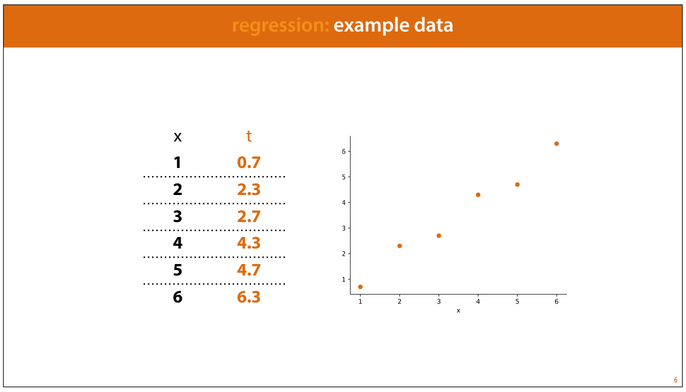
image source: **https://allisonhorst.github.io/palmerpenguins/**



feature space

flipper length (dm)	body mass (kg)
1.93	3.65
1.88	4.70
1.90	4.95
2.17	5.70
1.90	3.32
1.92	5.70
2.23	5.00
2.05	3.45
2.08	3.05
1.93	5.30

As we saw, the **linear regression model** is simply a linear function that maps the feature(s) to the target value. In the case of one feature, such a function looks like a line. The only decision we have to make is *which line fits the data best*.

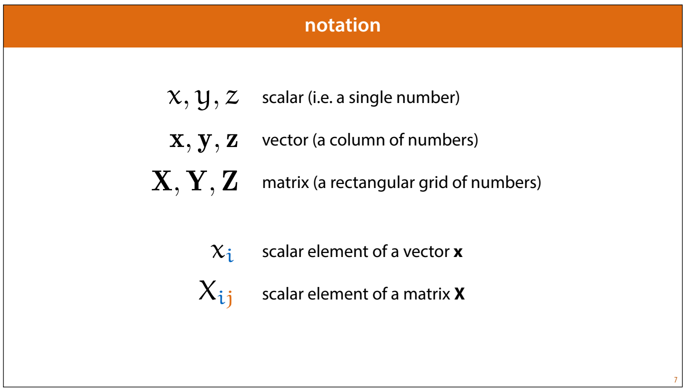


regression: example data

x	t
1	0.7
2	2.3
3	2.7
4	4.3
5	4.7
6	6.3

To simplify things we'll use this very simple data set in the rest of this lecture. There is one input feature **x**, one output value **t** (for target) and we have six instances.

We will assume that all our features are *numeric*. In a later lecture we will see how best to convert categoric features to numeric ones. We will develop linear regression for an arbitrary number of features m, but we will keep visualizing what we're doing on this one-feature dataset.

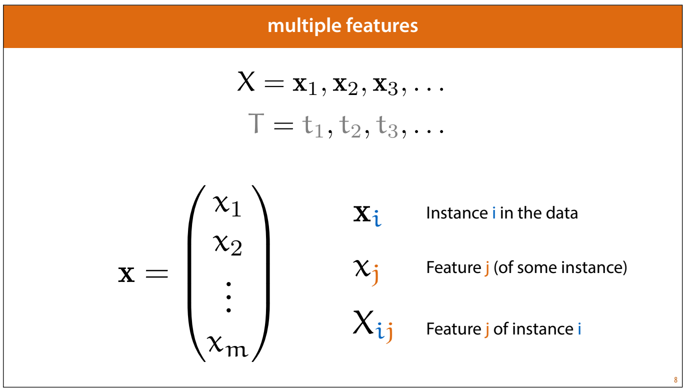


notation

x, y, z	scalar (i.e. a single number)
$\mathbf{x}, \mathbf{y}, \mathbf{z}$	vector (a column of numbers)
$\mathbf{X}, \mathbf{Y}, \mathbf{Z}$	matrix (a rectangular grid of numbers)
x_i	scalar element of a vector **x**
X_{ij}	scalar element of a matrix **X**

Throughout the course, we will use the following notation: lowercase non-bold for scalars, lowercase **bold** for vectors and uppercase bold for matrices.

When we're indexing *individual elements* of vectors and matrices, these are scalars, so they are non-bold.



multiple features

$$X = \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$$

$$T = t_1, t_2, t_3, \ldots$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

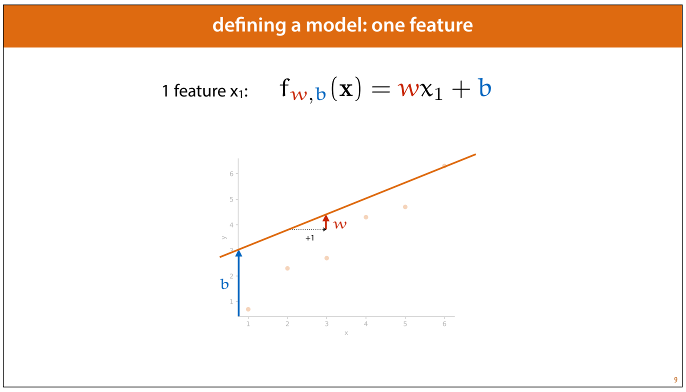
$\mathbf{x}_i$	Instance i in the data
x_j	Feature j (of some instance)
X_{ij}	Feature j of instance i

As we saw in the last lecture, an instance in machine learning is described by m *features* (with m fixed for a given dataset). We will represent this as a **vector** for each instance, with each element of the vector representing a feature.

This can be a little confusing, since we sometimes want to index the instance within the dataset and sometimes the features of a given instance. Pay attention to whether the letter we're indexing is bold or non-bold: a bold letter **x** with a subscript i refers to the i-th instance in the data (containinig all features). A non-bold letter x with an index i refers to the i-th scalar feature of some instance **x**.

In the rare cases where we need to refer to both the index of the instance, and the index of the feature within the instance, we will usually use an uppercase X. This makes sense if you imagine the data as a big matrix **X**, with the instances as rows, and the features as columns.

We'll occasionally deviate from this notation when doing so makes things clearer, but we'll point it out when that happens.



defining a model: one feature

1 feature x_1: $f_{w,b}(\mathbf{x}) = wx_1 + b$

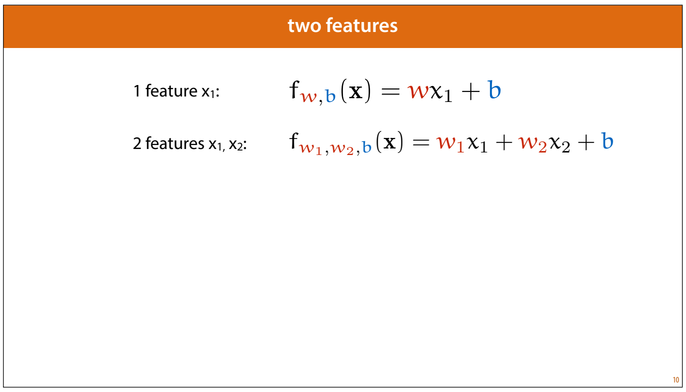
If we have one feature (as in this example) a standard linear regression model has two *parameters* (the numbers that determine which line we fit through our data): w the **weight** and b**,** the **bias**. The the weight is also sometimes called the *slope* and the bias is also sometimes called the *intercept*.

b determines where the line crosses the vertical axis. That is, what value f takes when x = 0.

w determines how much the line rises if we move one step to the right (i.e. increase x by 1)

For the line drawn here, we have b=3 and w=0.5.

Note that this isn't a very good fit for the data. Our job is to find better numbers w and b.

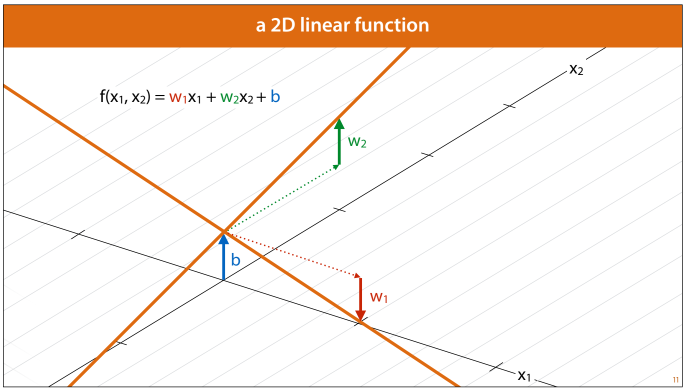


two features

1 feature x_1: $f_{w,b}(\mathbf{x}) = wx_1 + b$

2 features x_1, x_2: $f_{w_1,w_2,b}(\mathbf{x}) = w_1x_1 + w_2x_2 + b$

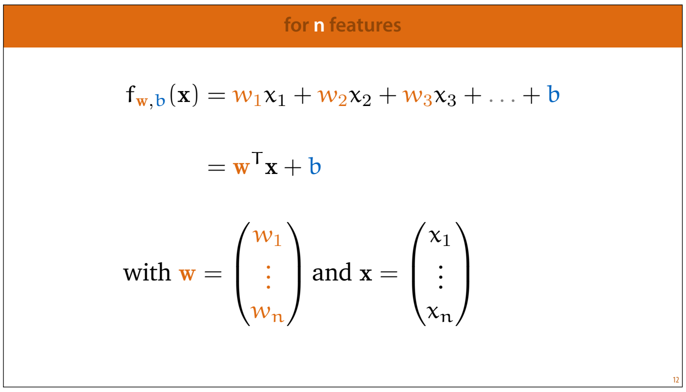
If we have multiple features, each feature gets its own **weight** (also known as a *coefficient*)



a 2D linear function

$f(x_1, x_2) = w_1x_1 + w_2x_2 + b$

Here's what that looks like. The thick orange lines together indicate a plane (which rises in the x_2 direction, and declines in the x_1 direction). The parameter b describes how high above the origin this plane lies (what the value of f is if *both features* are 0). The value w_1 indicates how much f increases if we take a step of 1 along the x_1 axis, and the value w_2 indicates how much f increases if we take a step of size 1 along the x_2 axis.



for n features

$$f_{\mathbf{w},b}(\mathbf{x}) = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + b$$

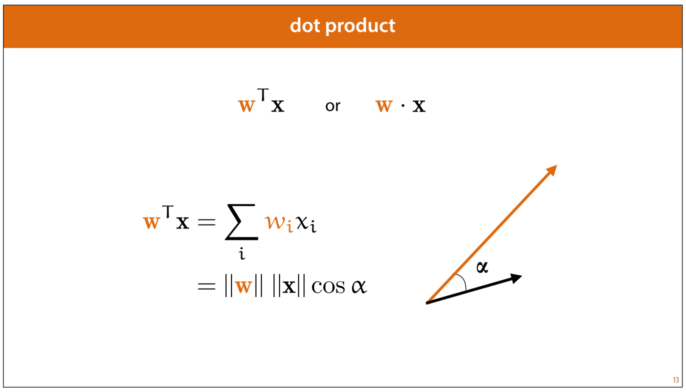
$$= \mathbf{w}^T\mathbf{x} + b$$

with $\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$ and $\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$

For an arbitrary number of features, the pattern continues as you’d expect. We summarize the w’s in a vector **w** with the same number of elements as **x**.

We call the w’s the weights, and b the bias. The weights and the bias are the *parameters* of the model. We need to choose these to fit the model to our data.

The operation of multiplying elements of **w** by the corresponding elements of **x** and summing them is the **dot product** of **w** and **x**.



dot product

$\mathbf{w}^T\mathbf{x}$ or $\mathbf{w} \cdot \mathbf{x}$

$$\mathbf{w}^T\mathbf{x} = \sum_i w_i x_i$$

$$= ||\mathbf{w}||\ ||\mathbf{x}|| \cos \alpha$$

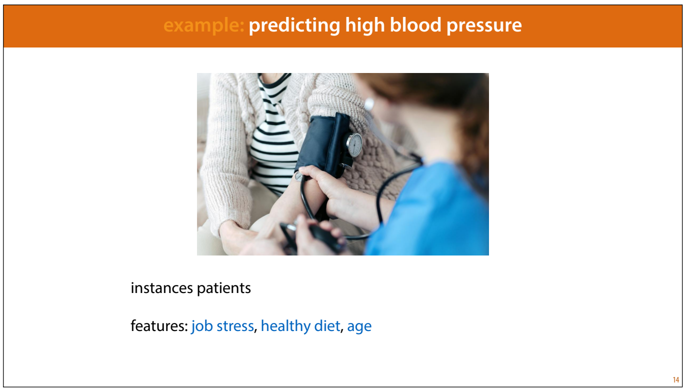
The **dot product** of two vectors is simply the sum of the products of their elements. If we place the features into a vector and the weights, then a linear function is simply their dot product (plus the b parameter).

The transpose (superscript T) notation arises from the fact that if we make one vector a row vector and one a column vector, and matrix-multiply them, the result is the dot product (try it).

The dot product also has a geometric interpretation: the dot product is equal to the lengths of the two vectors, multiplied by the cosine of the angle between them. We won't give you a proof, but we'll occasionally make use of this form of the dot product, so make sure you remember this.

The proof that the two definitions are equivalent is given in the preliminaries lecture.

The dot product will come back *a lot* in the rest of the course. We don't have time to discuss it in depth, but if your memory is hazy, we strongly recommend that you take a minute to go back to your linear algebra book and look up the various interpretations of what the dot product means.

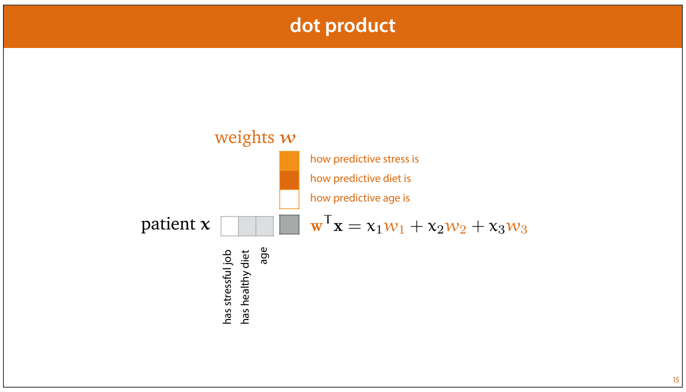


example: predicting high blood pressure

instances patients

features: job stress, healthy diet, age

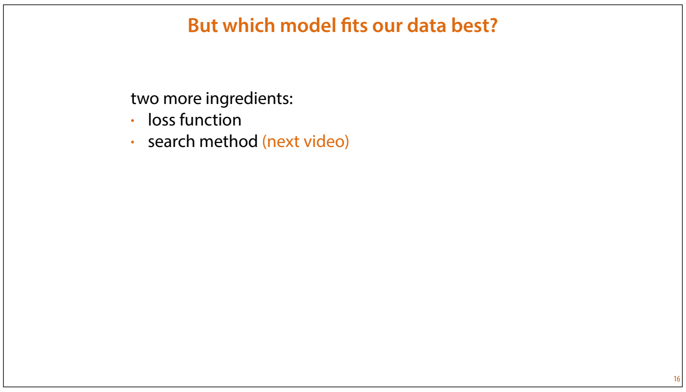
To build some intuition for the meaning of the weights **w**, let's look at an example. Imagine we are trying to predict the risk of high blood pressure based on these three features. We'll assume that the features are expressed in some number that measures these properties.



dot product

$$\mathbf{w}^T\mathbf{x} = x_1w_1 + x_2w_2 + x_3w_3$$

Here's what the dot product expresses. For some features, like job stress, we want to learn a positive weight (since more job stress should contribute to higher risk of high blood pressure). For others, we want to learn a negative weight (the healthier your diet, the lower your risk of high blood pressure). Finally, we can control the *magnitude* of the weights to control their relative importance: if age and job stress both contribute positively, but age is the bigger risk factor, we make both weights positive, but we make the weight for age bigger.



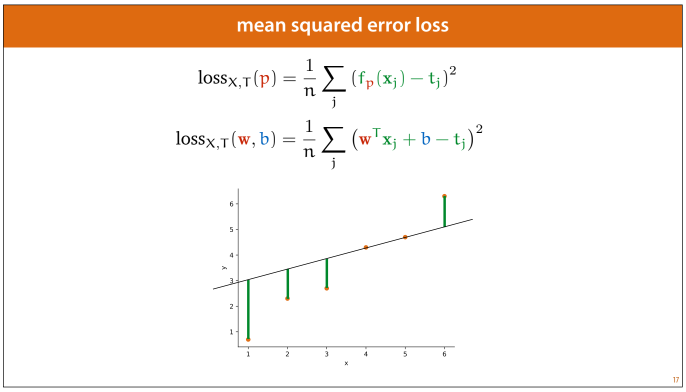
But which model fits our data best?

two more ingredients:

- loss function
- search method (next video)

So, that's our model defined in detail. But we still don't know *which* model to choose for a given dataset. Given some data, which values should we choose for the parameters **w** and b?

In order to answer this question, we need two more ingredients. First, we need a **loss function**, which tells us how well a particular choice of model does (for the given data) and second, we need a way to **search** the space of all models for a particular model that results in a low loss (a model for which the loss function returns a low value).



mean squared error loss

$$\text{loss}_{X,T}(p) = \frac{1}{n}\sum_j \left(f_p(\mathbf{x}_j) - t_j\right)^2$$

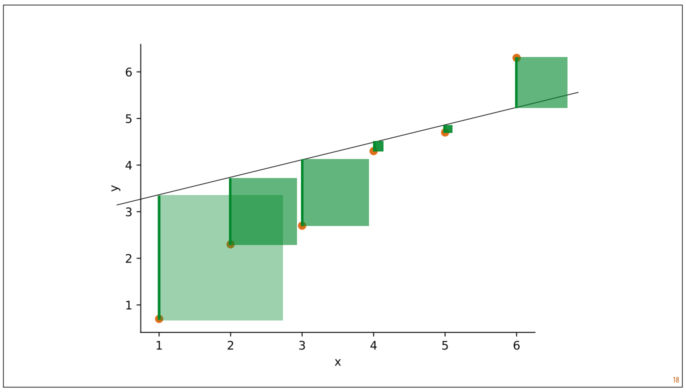
$$\text{loss}_{X,T}(\mathbf{w}, b) = \frac{1}{n}\sum_j \left(\mathbf{w}^T\mathbf{x}_j + b - t_j\right)^2$$

Here is a common loss function for regression: the **mean-squared error** (MSE) loss. We saw this briefly already in the previous lecture.

Note that the loss function takes a *model* as its argument. The model maps the data to the output, the loss function maps a model to a loss value. The data is a constant in the loss function.

The main thing a regression loss should do is to compare the model predictions to the actual values in our dataset, and return a large value if they are all very different, and a small value if they are all very close together. The difference between the prediction and the actual value is called the **residual**. We've drawn these here as green bars.

The MSE loss takes the residual for each instance in our data, squares them, and returns the average. One reason for the squaring step is to ensure that negative and positive residuals don't cancel out (giving us a small loss even though we have big residuals). But that's not the only reason.



The squares also ensure that big errors affect the loss more heavily than small errors. You can visualise this as shown here: the mean squared error is the mean of the areas of the green squares (it's also called *sum-of-squares loss*).

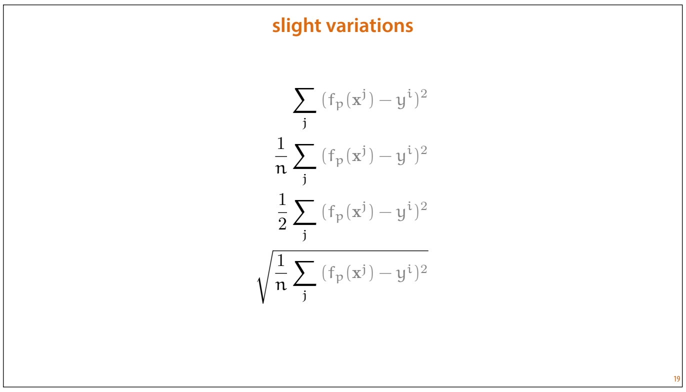
When we search for a well-fitting model, the search will try to reduce the big squares much more than the small squares.

If we think of the residuals as rubber bands, pulling on the regression line to pull it closer to the points, the rubber band on the bottom left pulls much harder than all the other ones. Therefore, any search algorithm trying to minimize this loss will be much more interested in moving the left of the line down than in moving the right of the line up.

It's not guaranteed that this is a good thing. Sometimes this behavior is desirable and sometimes it isn't. For now, this is just a simple loss function to get us started.

In later lectures, we will say more about when this kind of loss is appropriate and when it isn't. We will also see that this loss function follows from the assumption that our data contains noise coming from a *normal distribution*.

visualization stolen from **https://machinelearningflashcards.com/**



slight variations

$$\sum_j (f_p(x^j) - y^i)^2$$

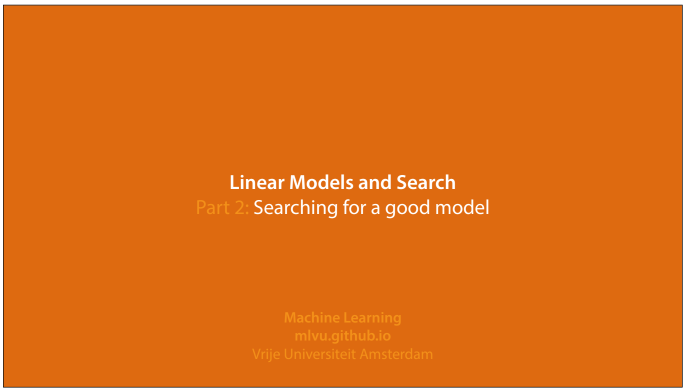
$$\frac{1}{n}\sum_j (f_p(x^j) - y^i)^2$$

$$\frac{1}{2}\sum_j (f_p(x^j) - y^i)^2$$

$$\sqrt{\frac{1}{n}\sum_j (f_p(x^j) - y^i)^2}$$

You may see slightly different versions of the MSE loss: sometimes we take the average of the squares, sometimes just the sum. Sometimes we multiply by 1/2 to make the derivative simpler. In practice, the differences don't mean much because we're not interested in the *absolute* value, just in how the loss changes from model to another.

We will switch between these based on what is most useful in a given context.

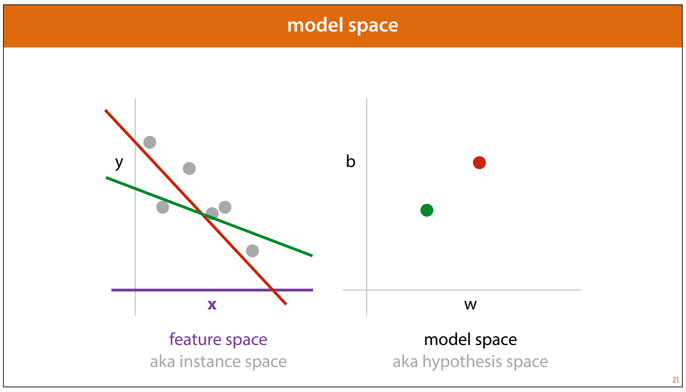


Linear Models and Search

Part 2: Searching for a good model

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

|section|Searching for a good model|
|video|https://surfdrive.surf.nl/files/index.php/s/wOMpuP3y06K5hc3/download|



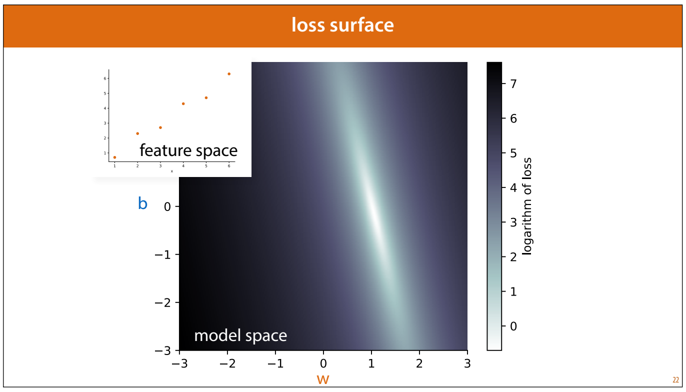
model space

feature space
aka instance space

model space
aka hypothesis space

Remember the two most important spaces of machine learning: the **feature space** and the **model space**. The loss function maps every point in the model space to a loss value.

In a single-feature regression problem plotted like this, the feature space is just the horizontal axis.



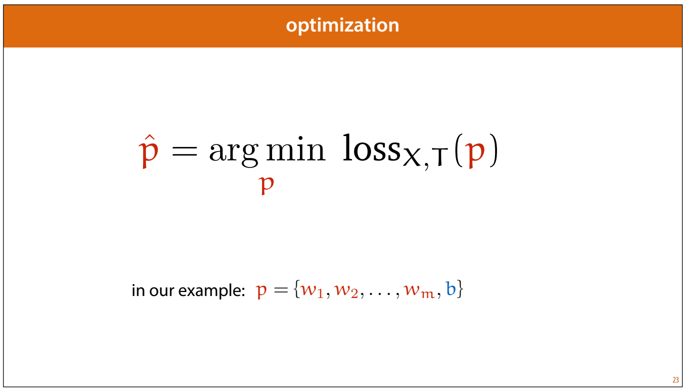
loss surface

As we saw in the previous lecture, we can plot the loss for every point in our model space. This is called the **loss surface** or sometimes the **loss landscape**. If you imagine a 2D model space, you can think of the loss surface as a landscape of rolling hills (or sometimes of jagged cliffs).

Here is what that actually looks like for the two parameters of the one-feature linear regression. Note that this is specific to the data we saw earlier. For a different dataset, we get a different loss landscape.

To minimize the loss, we need to search this space to find the brightest point in this picture. Or, the lowest point in the loss landscape. Remember that, normally, we may have hundreds of parameters so it isn't as easy as it looks. Any method we come up with, needs to work in any number of dimensions.

*We've plotted the logarithm of the loss as a trick to make this image visually easier to understand (it maps the values that are easy to tell apart to the values we care about). The logarithm is a monotonic function so log(loss(**w**, b)) has its minimum at the same place as loss(**w**, b).*



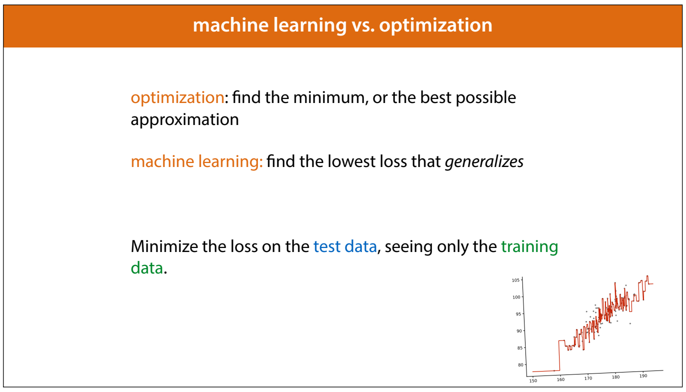
optimization

$$\hat{p} = \arg\min_{p}\ \text{loss}_{X,T}(p)$$

in our example: $p = \{w_1, w_2, \ldots, w_m, b\}$

The mathematical name for this sort of search is **optimization**. That is, we are trying to find the input (**p**, the model parameters) for which a particular function (the loss) is at its optimum (a maximum or minimum, in this case a minimum). Failing that, we'd like to find as low a value as possible.

We'll start by looking at some very simple approaches.



machine learning vs. optimization

optimization: find the minimum, or the best possible approximation

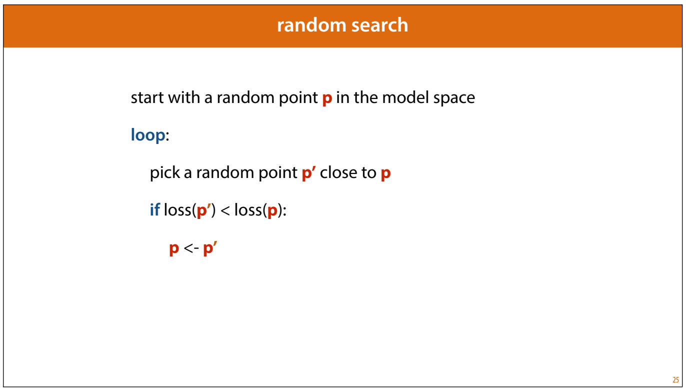
machine learning: find the lowest loss that *generalizes*

Minimize the loss on the test data, seeing only the training data.

We often frame machine learning as an optimization problem, and we use many techniques from optimization, but it’s important to recognize that there is a difference between optimization and machine learning.

Optimization is concerned with finding the absolute minimum (or maximum) of a function. The lower the better, with no ifs or buts. In **machine learning**, if we have a very expressive model class (like the regression tree from the last lecture), the model that actually minimizes the loss on the training data is the one that overfits. In such cases, we’re not looking to minimize the loss on the training data, since that would mean overfitting, we’re looking to minimize the loss on the *test data*. Of course, we don’t get to see the test data, so we use the training data as a stand in, and try to control against overfitting as best we can.

In the case of underpowered models like the linear model, this distinction isn’t too important, since they’re very unlikely to overfit. Here, the model that minimizes the loss on the training data is likely the model that minimizes the loss on the test data as well. For now, we'll just try some simple optimization algorithms to find the absolute minimum of the loss, and worry about overfitting later.



random search

```
start with a random point p in the model space
loop:
    pick a random point p′ close to p
    if loss(p′) < loss(p):
        p <- p′
```

Let’s start with a very simple example: **random search**. We simply make a small step to a nearby point. If the loss goes up, we move back to our previous point. If it goes down we stay in the new point. Then we repeat the process.

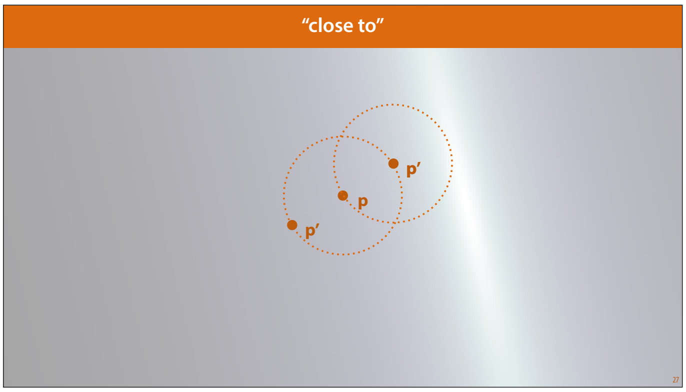
We usually stop the loop when the loss gets to a pre-defined level, or we just run it for a fixed number of iterations, and we see how well we've done.

analogy: hiker in a snowstorm

A common analogy is a *hiker in a snowstorm*. Imagine you're hiking in the mountains, and you're caught in a snowstorm. You can't see a thing, and you'd like to get down to your hotel in the valley, or failing that, you'd like to get to as low a point as possible.

You take a step in a random direction. If you're moving up, you step back to where you came from, if you're moving down, you repeat the process with a new random direction. This is, in effect, what random search is doing. More importantly, it's how *blind* random search is to the larger structure of the landscape. It can only see what's right in front of it.

image source: https://www.wbur.org/hereandnow/2016/12/19/rescue-algonquin-mountain

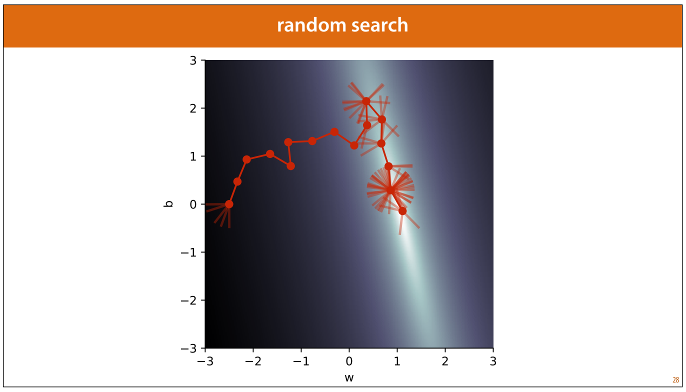


"close to"

To implement the random search we need to define how to pick a point "close to" another in model space.

One simple option is to choose the next point by sampling uniformly among all points with some pre-chosen distance r from the current point.

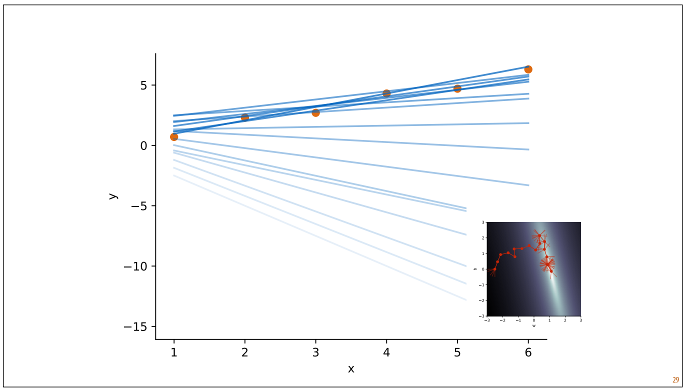
Put more formally: we pick from the hypersphere (or circle, in 2D) with radius r, centered on ***p****.*



random search

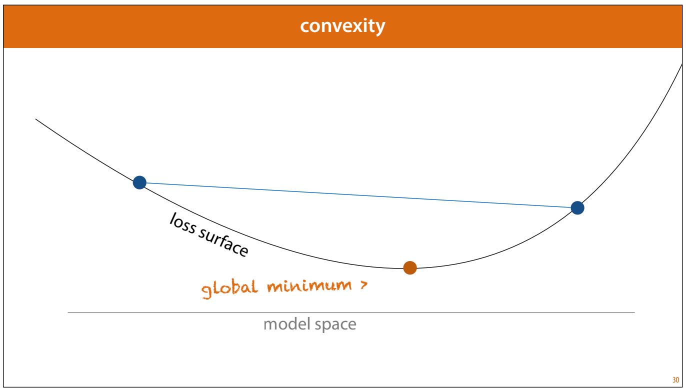
Here is random search in action. The transparent red offshoots are steps that turned out to be worse than the current point (steps that went uphill). The algorithm starts on the left, and slowly (with a bit of a detour) stumbles in the direction of the low loss region.

As we can see, it doesn't exactly make a beeline for the lowest point, but it gets there eventually.



Here is what it looks like in **feature space**. The first model (bottom-most line) is entirely wrong, and the search slowly moves, step by step, towards a reasonable fit on the data.

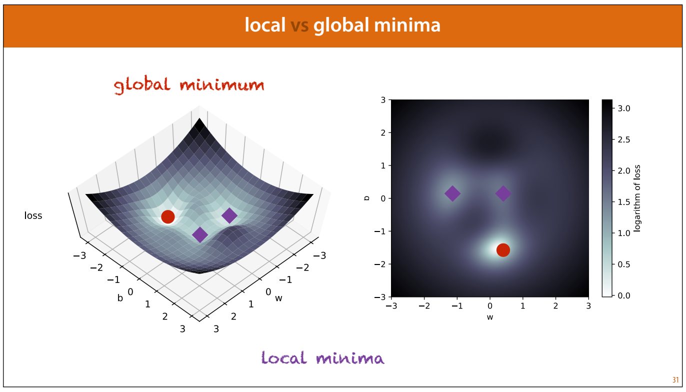
Every blue line in this image corresponds to a red dot in the model space (inset).



convexity

One of the reasons such a simple approach works well enough here is that our problem is **convex**. A surface (like our loss landscape) is convex if a line drawn between any two points on the surface lies entirely above the surface. One of the implications of convexity is that any point that looks like a minimum locally (because all nearby points are higher), it must be the **global minimum**: it's lower than any other point on the surface.

This means that so long as we know we're moving down (to a point with lower loss), we can be sure we're moving towards the global minimum: the best of all possible models.

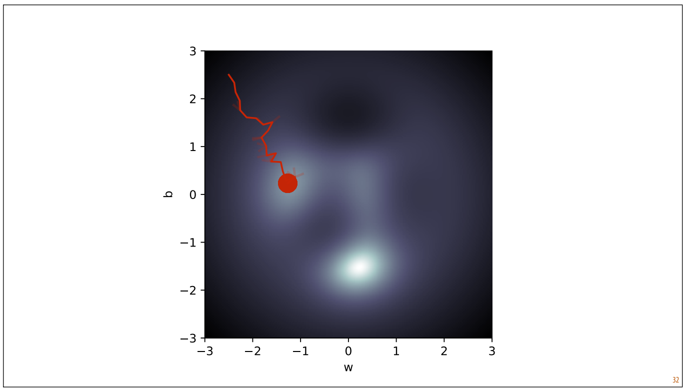


local vs global minima

Let's look at what happens if the loss surface *isn't* convex: what if the loss surface has multiple **local minima**? These are points that are lower than all nearby points, but if we move far enough away from them, we *can* find a point that is even lower.

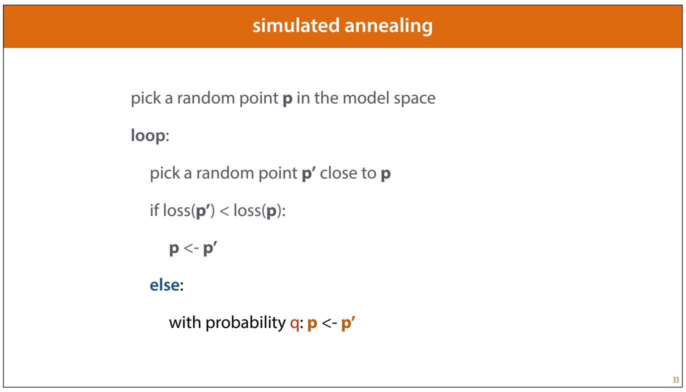
Here's a loss surface with a more complex structure. The two purple diamonds are the lowest point in their respective *neighborhoods*, but the red disc is the lowest point globally.

This loss surface isn't based on actual data. It's just some function that illustrates the idea.



Here we see random search on our more complex loss surface. As you can see, it heads quickly for one of the local minima, and then gets stuck there. No matter how many more iterations we give it, it will never escape.

Note that changing the step size will not help us here. Once the search is stuck, it stays stuck.



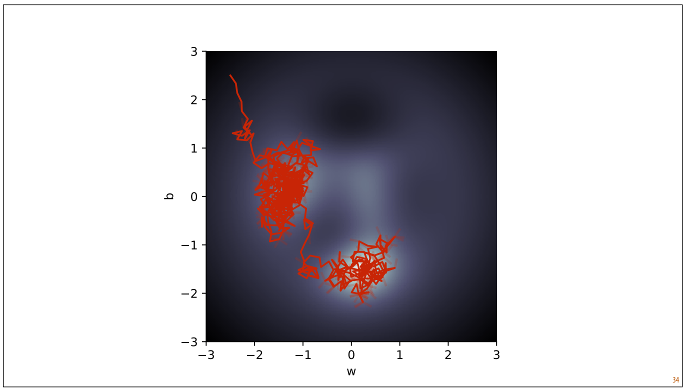
simulated annealing

```
pick a random point **p** in the model space
loop:
    pick a random point **p'** close to **p**
    if loss(**p'**) < loss(**p**):
        **p** <- **p'**
    else:
        with probability q: **p** <- **p'**
```

There are a few tricks that can help us to escape local minima. Here's a popular one, called **simulated annealing**: if the next point chosen isn't better than the current one, we still pick it, but only with some small probability. In other words, we allow the algorithm to *occasionally* travel uphill.

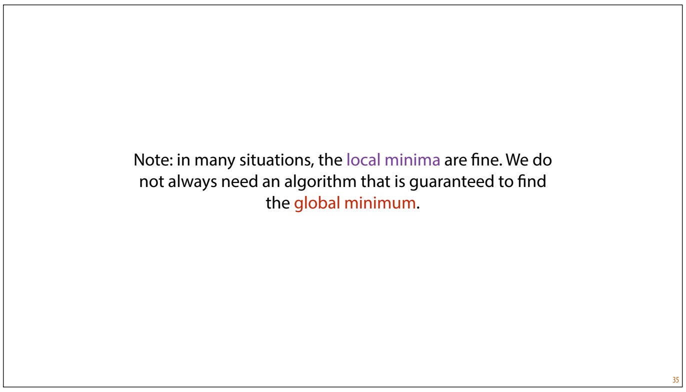
This means that whenever it gets stuck in a local minimum, it still has some probability of escaping, and finding the global minimum.

The name "simulated annealing" is a bit of a historical accident, so don't read too much in to it. It comes from the fact that this algorithm can be used to simulate the cooling of a material like metal. The carefully controlled cooling of a material to promote the growth of particular kinds of crystals is called annealing. In physical terms this is like looking for the minimum in an energy landscape, which is mathematically similar to our loss landscape.



Here is a run of simulated annealing on our non-convex problem. We see that it still hits the local minimum first, but after a while it manages to jump out, and to find the global minimum.

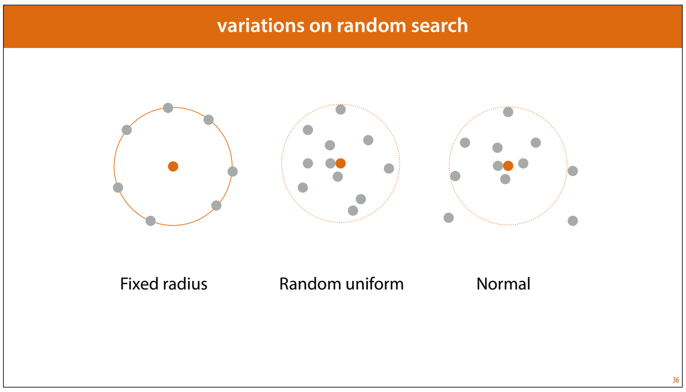
Of course, with this algorithm, there is always the possibility that it will jump out of the global minimum again and move to a worse minimum. That shouldn't worry us, however, so long as we remember the best model we've observed over the entire run. Then we can just let simulated annealing jump around the model space driven partly by random noise, and partly by the loss surface.



Note: in many situations, the local minima are fine. We do not always need an algorithm that is guaranteed to find the global minimum.

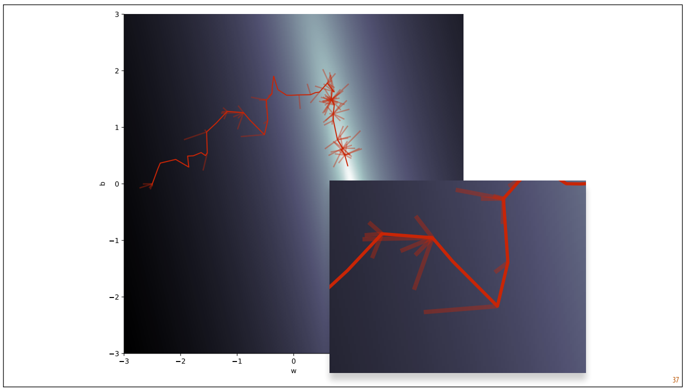
All this talk about global minima may suggest that the local minima are always terrible. Remember, however that if we have a complex model, the global minimum will probably overfit. In such cases, we may actually be more interested in finding a good local minimum.

In short, we want to think carefully about whether our algorithm can escape *bad* local minima, but that doesn't mean that local minima are always bad solutions.

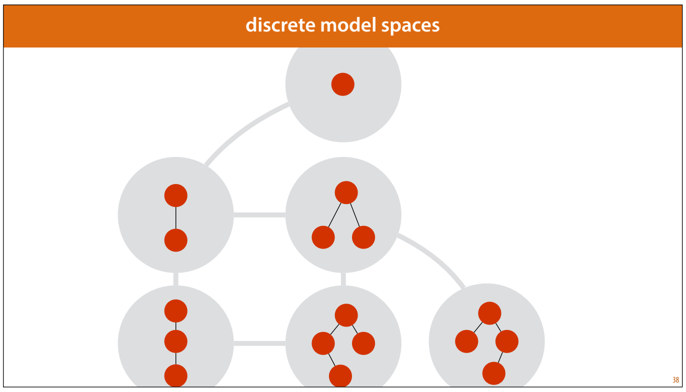


variations on random search

The fixed step size we used so far is just one way to sample the next point. To allow the algorithm to occasionally make smaller steps, you can sample **p'** so that it is *at most* some distance away from **p**, instead of *exactly*. Another approach is to sample the distance from **a Normal distribution**. That way, most points will be close to the original **p**, but every point in the model space can theoretically be reached in one step.



Here is what random search looks when the steps are sampled from a normal distribution. Note that the “failed” steps all have different sizes.



discrete model spaces

The space of linear models is **continuous**: between every two models, there is always another model, no matter how close they are together. *

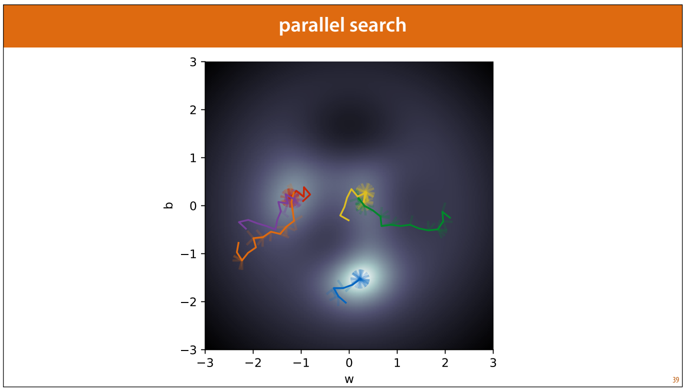
The alternative is a **discrete model space**. For instance, the space of all trees is discrete. If our model takes the form of a tree (like the decision tree we saw in the last lecture), then we don't always have another model "in between" any two given models. In this case, some search algorithms no longer work, but random search and simulated annealing can still be used.

You just need to define which models are “close” to each other. In this slide, we've decided that two trees are close if I can turn one into the other by adding or removing a single node.

Random search and simulated annealing can now be used to search this space to find the tree model that gives the best performance.

In practice, we usually use a different method to search for decision trees and regression trees, which we will introduce in a later lecture. The point here is just that if you are searching a discrete space, random search and simulated annealing still work.

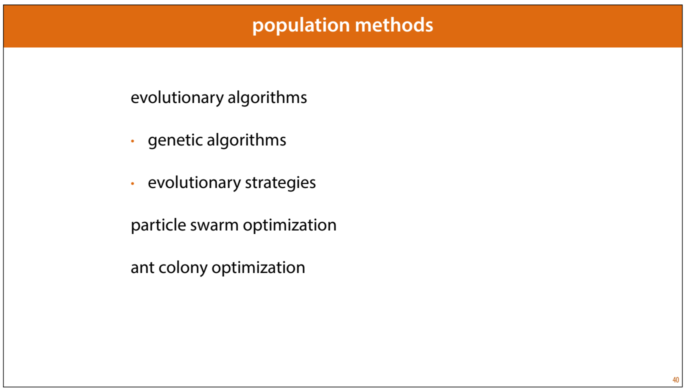
** Strictly speaking, this is not a complete definition of a continuous space, but it's the property that matters for us.*



parallel search

Another thing you can do is just to run random search a couple of times independently (one after the other, or in parallel). If you’re lucky one of these runs may start you off close enough to the global minimum.

For simulated annealing, doing multiple runs makes less sense. We can show that there’s not much difference between 10 runs of 100 iterations and one run of 1000. The only reason to do multiple runs of simulated annealing is because it’s easier to parallelize over multiple cores or machines.



population methods

evolutionary algorithms

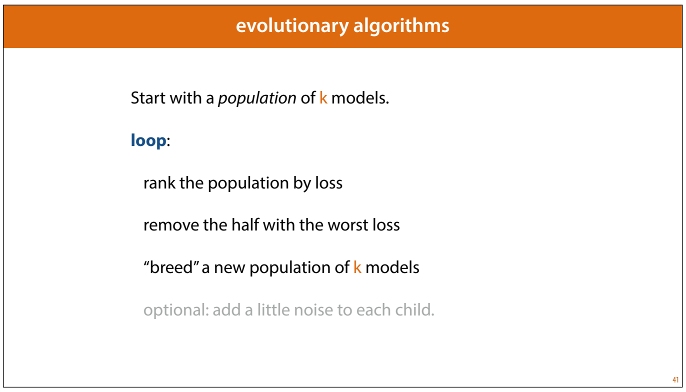
- genetic algorithms
- evolutionary strategies

particle swarm optimization

ant colony optimization

To make parallel search even more useful, we can introduce some form of communication or synchronization between the searches happening in parallel. If we see the parallel searches as a *population* of agents that occasionally “communicate” in some way, we can guide the search a lot more. Here are some examples of such **population methods**. we won’t go into this too deeply. We will only take a (very) brief look at evolutionary algorithms.

Often, there are specific variants for discrete and for continuous model spaces.



evolutionary algorithms

Start with a *population* of k models.

loop:

rank the population by loss

remove the half with the worst loss

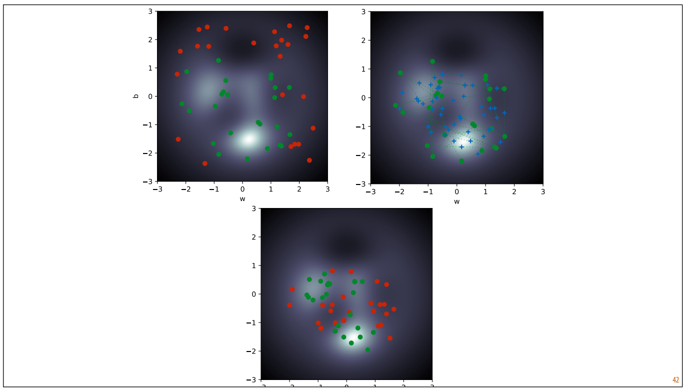
“breed” a new population of k models

optional: add a little noise to each child.

Here is a basic outline of an evolutionary method (although many other variations exist). We start with a population of models, we remove the half with the worst loss, and pair up the remainder to breed a new population.

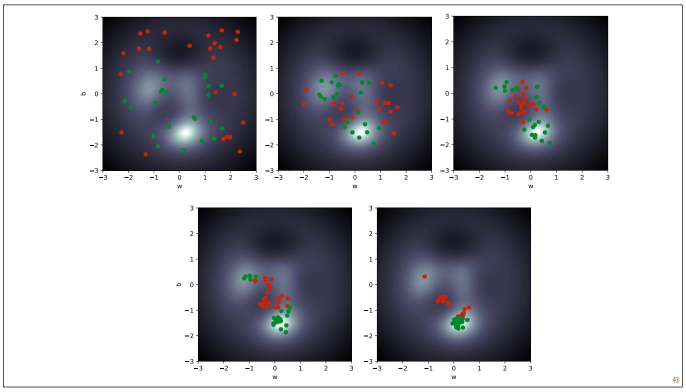
In order to instantiate this, we need to define what it means to “breed” a population of new models from an existing population. A common approach is to select to random parents and to somehow average their models. This is easy to do in a continuous model space: we can literally average the two parent models to create a child.

In a discrete model space, it’s more difficult, and it depends more on the specifics of the model space. In such case, designing the breeding process (sometimes called the **crossover operator**) is usually the most difficult part of designing an effective evolutionary algorithm.

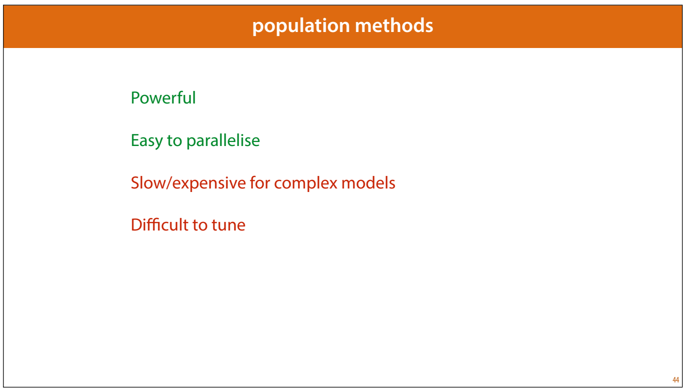


Here’s what a very basic evolutionary search looks like on our non-convex loss surface. We start with a population of 50 models, and compute the loss for each. We kill the worst 50% (the red dots) and keep the best 50% (the green dots).

We then create a new population (the blue crosses), by randomly pairing up parents from the green population, and taking the point halfway between the two parents, with a little noise added. Finally, we take the blue crosses as the new population and repeat the process.



Here are five iterations of the algorithm. Note that in the intermediate stages, the population covers both the local and the global minima.



population methods

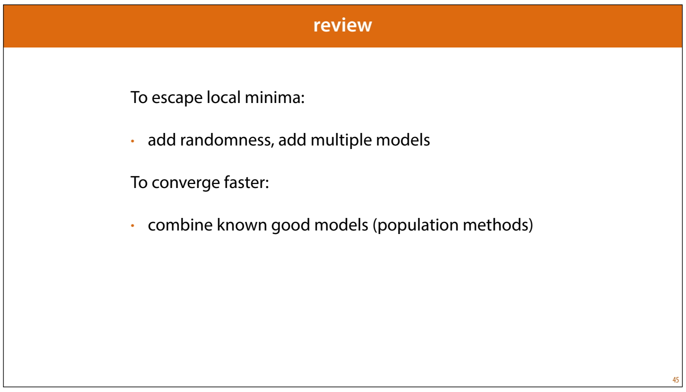
Powerful

Easy to parallelise

Slow/expensive for complex models

Difficult to tune

Population methods are very powerful, but computing the loss for so many different models is often expensive. They can also come with a lot of different parameters to control the search, each of which you will need to carefully tune.



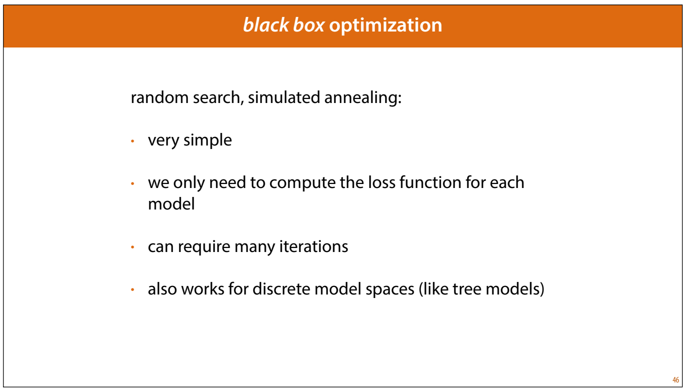
review

To escape local minima:

- add randomness, add multiple models

To converge faster:

- combine known good models (population methods)



black box optimization

random search, simulated annealing:

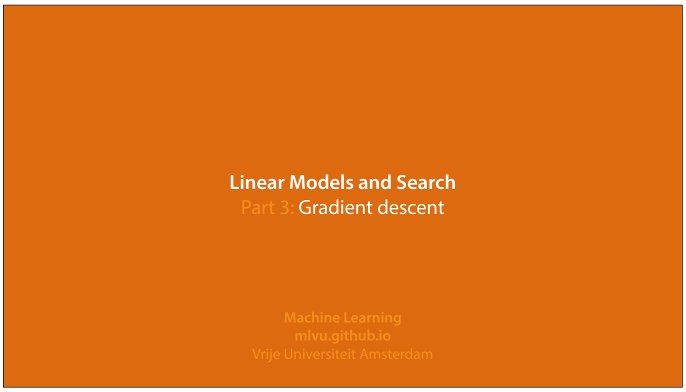
- very simple
- we only need to compute the loss function for each model
- can require many iterations
- also works for discrete model spaces (like tree models)

46

All these search methods are instances of **black box optimization**.

Black box optimization refers to those methods that only require us to be able to compute the loss function. We don’t need to know anything about the internals of the model. These are usually very simple starting points. Often, there is some knowledge about your model that you can add to improve the search, but sometimes the black box approach is good enough. If nothing else, they serve as a good starting point and point of comparison for the more sophisticated approaches.

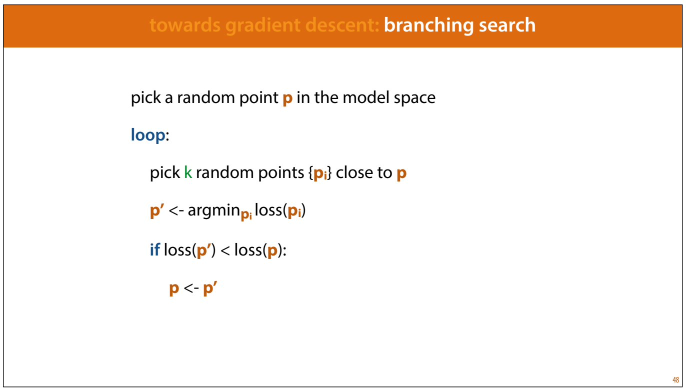
In the next video we’ll look at a way to improve the search by opening up the black box for continuous models: gradient descent.



Linear Models and Search
Part 3: Gradient descent

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

|section|Gradient descent|
|video|https://surfdrive.surf.nl/files/index.php/s/0WVzpZvce6RqVl4/download|



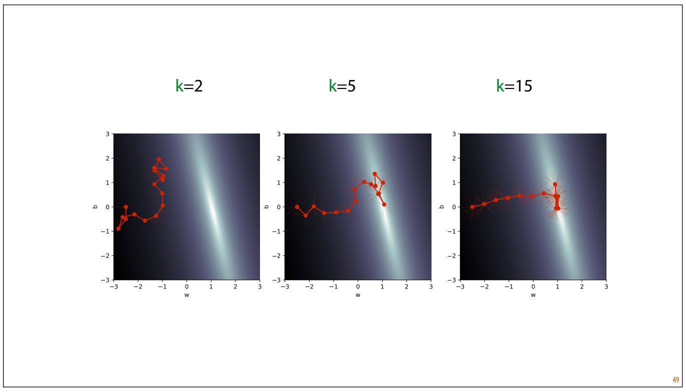
towards gradient descent: branching search

```
pick a random point $p$ in the model space
loop:
    pick $k$ random points $\{p_i\}$ close to $p$
    $p'$ <- $\text{argmin}_{p_i}$ loss($p_i$)
    if loss($p'$) < loss($p$):
        $p$ <- $p'$
```

48

As a stepping stone to what we’ll discuss in this video, let’s take the random search from the previous video, and add a little more inspection of the local neighborhood before taking a step. Instead of taking one random step, we’ll look at k random steps and move in the direction of the one that gives us the lowest loss.

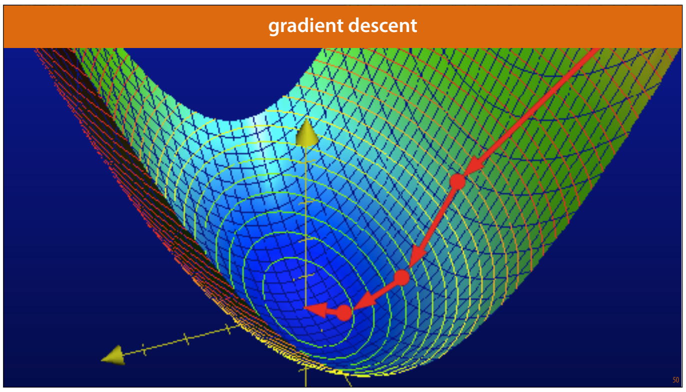
In the hiker analogy, you can think of this algorithm as the case where the hiker taps his foot on the ground in a couple of random directions, and then moves in the direction with the strongest downward slope.



Here's what that looks like for a few values of k.

As you can see, the more samples we take, the more directly we head for the region of low loss. The more closely we inspect our local neighbourhood, to determine in which direction the function decreases quickest, the faster we converge.

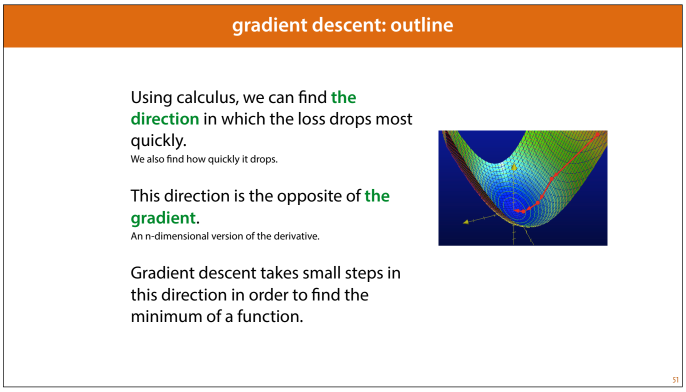
The lesson here is that the better we know in which direction the loss decreases, the faster our search converges. In this case we pay a steep price: we have to evaluate our function 15 times to work out a better direction.



gradient descent

However, if our model space is continuous, and if our loss function is smooth, we don't *need* to take multiple samples to guess the direction of fastest descent: *we can simply derive it, using calculus*. This is the basis of the **gradient descent algorithm**.

image source: **http://charlesfranzen.com/posts/multiple-regression-in-python-gradient-descent/**



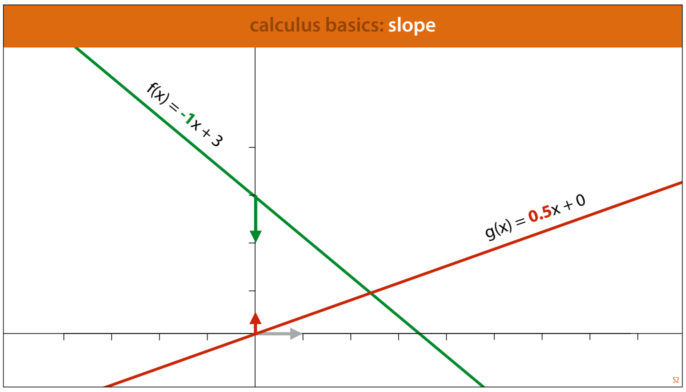
gradient descent: outline

Using calculus, we can find the direction in which the loss drops most quickly.
We also find how quickly it drops.

This direction is the opposite of the gradient.
An n-dimensional version of the derivative.

Gradient descent takes small steps in this direction in order to find the minimum of a function.

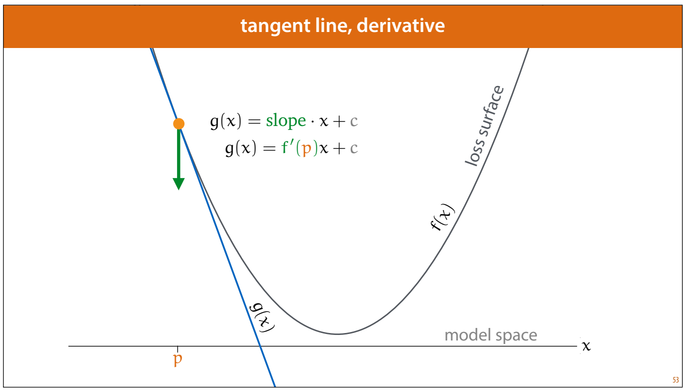
The idea of gradient descent is relatively simple, but it's easy to get blinded by the mathematical notation. Here are the main ideas to keep in mind.



calculus basics: slope

Before we dig in to the gradient descent algorithm, let's review some basic principles from calculus. First up, **slope**. The slope of a linear function is simply **how much it moves up** if we move one step to the right. In the case of f(x) in this picture, the slope is *negative*, because the line moves down.

In our 1D regression model, the parameter w was the slope. In this case, however we will investigate the slope of a linear function approximation the loss landscape, not of the model (be sure not to confuse these two).



tangent line, derivative

$$g(x) = \text{slope} \cdot x + c$$

$$g(x) = f'(p)x + c$$

The **tangent line** of a function at particular point p is the line that just touches the function at **x** without crossing it. The tangent line is a kind of **approximatio**n to our function. So long as we stay close to p, the function f(x) and the tangent line g(x) behave very similarly.

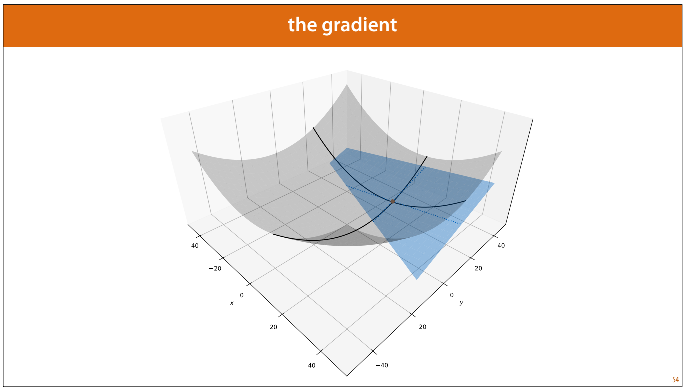
This is where the **derivative** f'(x) comes in. The **derivative of a function gives us the slope of the tangent line**. Since the slope tells us how quickly a function rises, and the tangent line is an approximation to f(x) at p, the slope of the tangent line tells us how quickly f(x) rises around the point p.

Traditionally, we find the minimum of a function by setting the derivative equal to 0 and solving for x. This gives us the point where the tangent line has slope 0, and is therefore horizontal.

For complex models, it may not be possible to solve for x in this way. However, we can still use the derivative to *search* for the minimum. Looking at the example in the slide, we note that the tangent line moves down (i.e. the slope is negative). This tells us that we should move to the right to follow the function downward. As we take small steps to the right, the derivative stays negative, but gets smaller and smaller as we close in on the minimum. This suggests that the *magnitude* of the slope lets us know how big the steps are that we should take. That is, if the slope of the tangent line is big, the function is dropping quickly, and we can take a big step. If the slope of the tangent line is small, the function is dropping more slowly, and we might be getting closer to the minimum.

This assumes that a function doesn't suddenly change from dropping very steeply to flattening out. That's not always true, but it's a reasonable hypothesis to start from. As we will see later, gradient descent can always correct if the assumption is wrong.

The first thing we need to do, is to extend this idea to functions of multiple inputs.

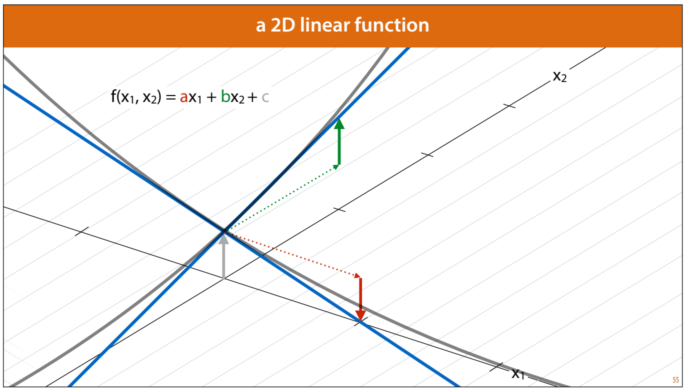


the gradient

To apply this principle in machine learning, we'll need to generalize it for loss functions with multiple inputs (i.e. for models with multiple parameters). We do this by generalizing the tangent line to a tangent (hyper)plane. The derivative then becomes a **gradient** vector that describes the way this hyperplane is angled.

Once we have this hyperplane, we can use it to work out in which direction the function grows and shrinks the quickest. As in the one-dimensional case, the tangent hyperplane is a **local approximation of the function**. Zoomed out like this, the hyperplane and the function look nothing alike, but if we zoom in close enough on the point where they touch, they behave almost exactly the same.

This is useful, because in a hyperplane it's very easy to see in which direction it goes down the quickest. Much easier than it is for a complicated beast like our loss function itself. Since the hyperplane approximates the loss function, this is also the direction in which the loss decreases the quickest. At least, so long as we don't move away too far from the neighborhood where the hyperplane is a good approximation of the loss function.



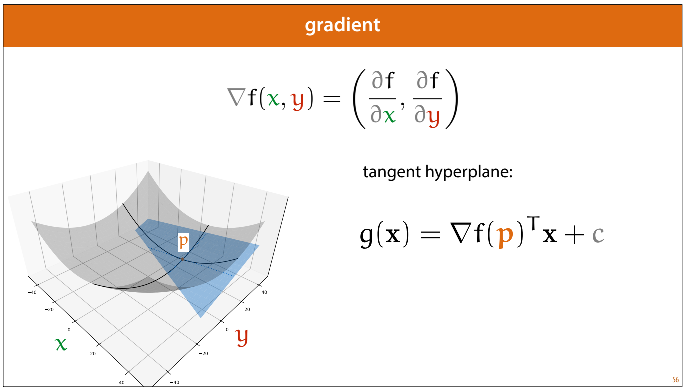
a 2D linear function

$f(x_1, x_2) = ax_1 + bx_2 + c$

Remember, that this is how we express a linear function in n dimensions: we assign each dimension a slope, and add a single bias (c).

In this image, the two weights of the linear function (a and b) are just one slope per dimension. If we move one step in the direction of x_1, we move up by a, and if we move one step in the direction of x_2, we move up by b.

This is the same picture we saw earlier for the linear function, but we're using it in a different way. Earlier, it represented a model with two features and three parameters. Here, it will serve as a linear approximation for a loss surface for a model with two parameters. That is, the plane x1, x2 is our model space, not our feature space, and the hyperplane is not a single model, but a stand-in for our loss function.



gradient

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

tangent hyperplane:

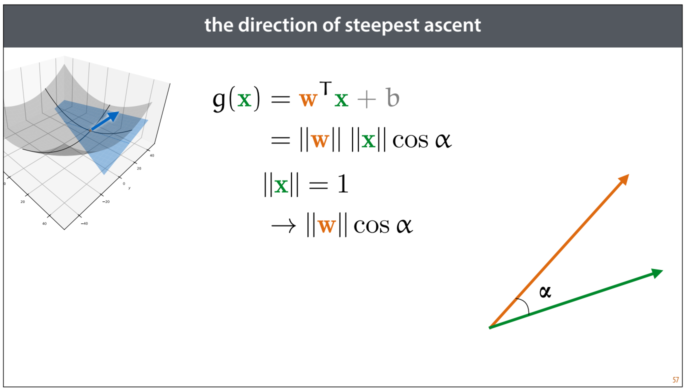
$$g(\mathbf{x}) = \nabla f(\mathbf{p})^T \mathbf{x} + c$$

We are now ready to show haw the gradient can be worked out. Any function from n inputs to one output has n variables for which we can take the derivative. These are called **partial derivatives**: they work the same way as regular derivatives, except that you when you take the derivative with respect to one variable x, you treat the other variables (y) as constants.

One thing that is sometimes a little confusing is that the gradient of a function f(·) is often written as another function ∇f(·). This ∇f(·) tells us not what the gradient is at a specific point but for all points. This is the same with the derivative: at a particular point, the derivative is some numerical value, but over all points, the derivative of f(x) is another function f'(x). If we take the gradient to be a function like this, then the tangent hyperplane of f(**x**) *at point **p*** is the function $g(\mathbf{x}) = \nabla f(\mathbf{p})^T\mathbf{x} + c$.

It is on this linear function, g(**x**) that we want to work out the direction of steepest ascent. The answer will be that the gradient ∇f(**p**) points exactly in that direction.

The gradient is sometimes defined as a row vector, and sometimes as a column vector. In machine learning contexts, the latter usually makes most sense.



the direction of steepest ascent

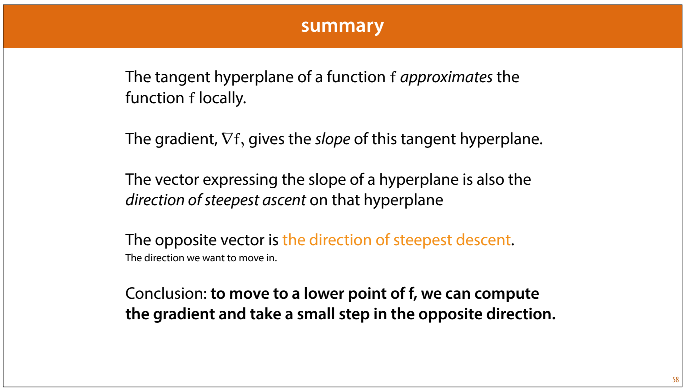
$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$
$$= \|\mathbf{w}\|\,\|\mathbf{x}\|\cos\alpha$$
$$\|\mathbf{x}\| = 1$$
$$\to \|\mathbf{w}\|\cos\alpha$$

To make this clear, we will write **w** = ∇f(**p**), so that g(**x**) looks like a plain old linear function. All we want to show is that w is the direction in which this function grows the quickest.

Since g(**x**) is linear, many details don't matter: we can set the bias b to zero, since that just translates the hyperplane up or down. Next, It doesn't matter *how big* a step we take in any direction, so we'll take a step of size 1. Finally, it doesn't matter where we start from, so we will just start from the origin. So the question becomes: for which input **x** of magnitude 1 (which unit vector) does g(**x**) provide the biggest output?

To see the answer, we need to use the geometric definition of the dot product. Since we required that ||**x**||= 1**,** this disappears from the equation, and we only need to maximize the quantity ||**w**|| cos(α) (where only α depends on our choice of **x**, and **w** is the gradient we computed). cos(α) is maximal when α is zero: that is, when **x** and **w** are pointing in the same direction.

In short: w, the gradient, *is* the direction of steepest ascent. This means that -**w** is the direction of steepest *descent*.



summary

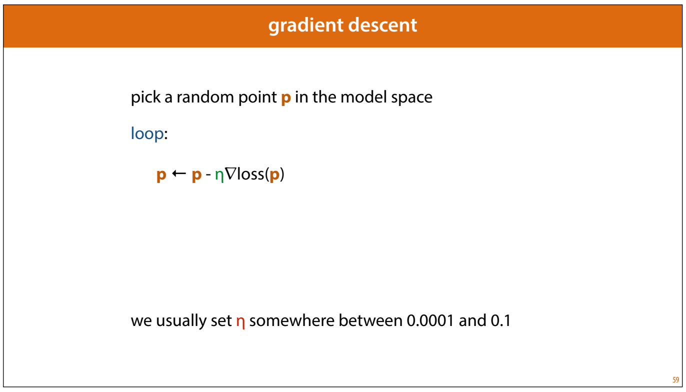
The tangent hyperplane of a function f *approximates* the function f locally.

The gradient, ∇f, gives the *slope* of this tangent hyperplane.

The vector expressing the slope of a hyperplane is also the *direction of steepest ascent* on that hyperplane

The opposite vector is the direction of steepest descent.
The direction we want to move in.

Conclusion: **to move to a lower point of f, we can compute the gradient and take a small step in the opposite direction.**



gradient descent

```
pick a random point p in the model space
loop:
    p ← p - η∇loss(p)
```

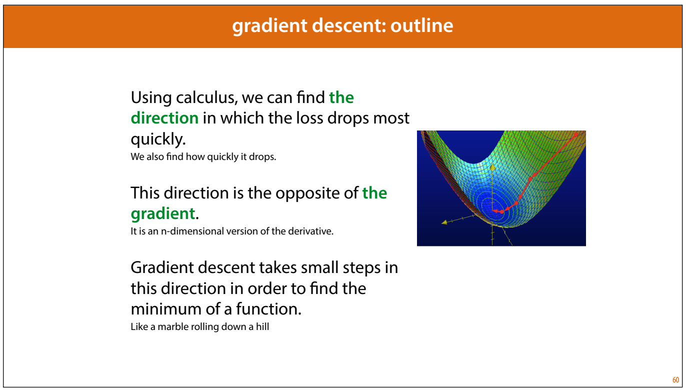
we usually set η somewhere between 0.0001 and 0.1

Here is the **gradient descent algorithm**. Starting from some candidate **p**, we simply compute the gradient at **p**, subtract it from the current choice, and iterate this process:

- We *subtract*, because the gradient points *uphill*. Since the gradient is the direction of steepest ascent, the negative gradient is the direction of steepest *des*cent.
- Since the gradient is only a *local approximation* to our loss function, the bigger our step, the more we go wrong because the approximation is incorrect. Usually, we scale down the step size indicated by the gradient by multiplying it by a value η (eta), called the **learning rate**. This value is chosen by trial and error, and remains constant throughout the search (at least in the simplest version of the algorithm).

Note again a potential point of confusion: we have two linear functions here. One is the *model*, whose parameters are indicated by **w** and b. The other is the tangent hyperplane to the loss function, whose slope is indicated by ∇loss(**p**) here. These are different functions on different spaces.

We can iterate for a fixed number of iterations, until the loss gets low enough, or until the gradient gets close enough to the zero vector, which implies we've reached a local minimum.



gradient descent: outline

Using calculus, we can find **the direction** in which the loss drops most quickly.

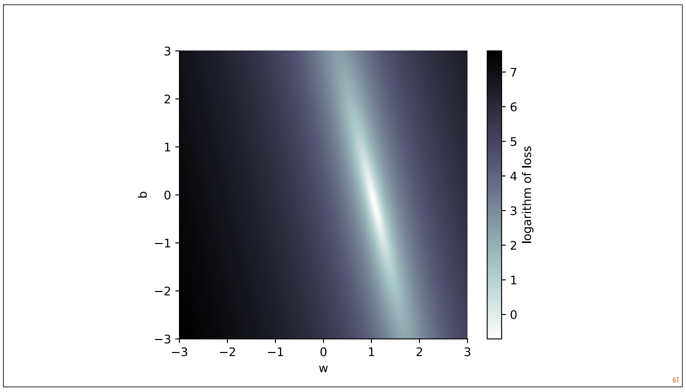
We also find how quickly it drops.

This direction is the opposite of **the gradient**.

It is an n-dimensional version of the derivative.

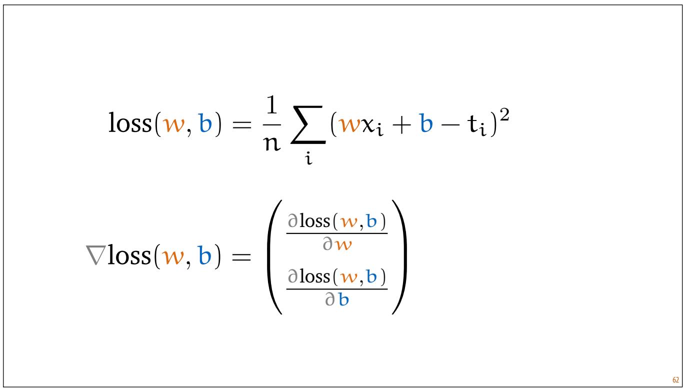
Gradient descent takes small steps in this direction in order to find the minimum of a function.

Like a marble rolling down a hill



Let’s go back to our example problem, and see how we can apply gradient descent here.

Unlike random search, it’s not enough to just compute the loss for a given model, **we need the gradient of the loss**. We'll start by working this out.

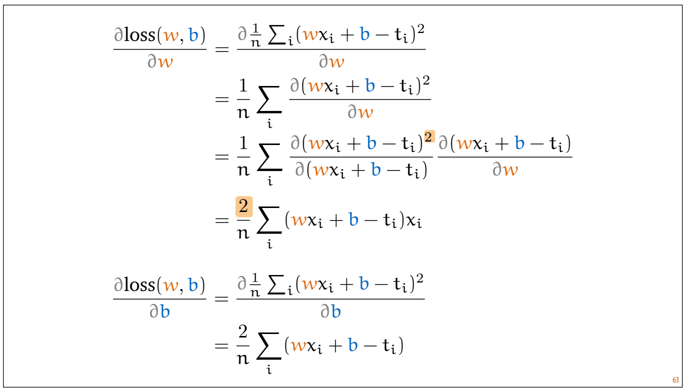


$$\text{loss}(w, b) = \frac{1}{n}\sum_i (wx_i + b - t_i)^2$$

$$\nabla\text{loss}(w, b) = \begin{pmatrix} \frac{\partial\text{loss}(w,b)}{\partial w} \\ \frac{\partial\text{loss}(w,b)}{\partial b} \end{pmatrix}$$

Here is our loss function again. The gradient is just a vector of all the partial derivatives we can take for it: one for the parameter w and one for the parameter b.

To simplify the notation we've let x_i refer to the only feature of instance i.



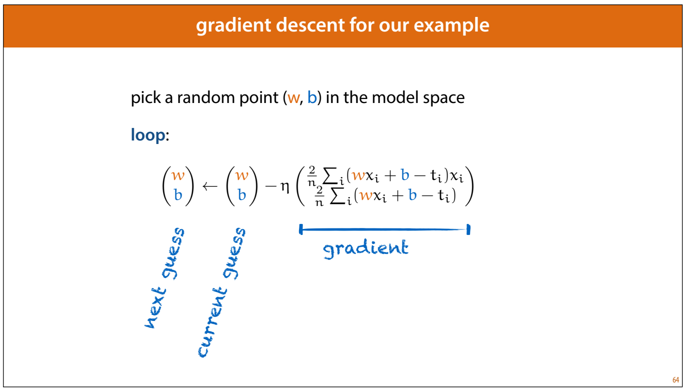
$$\begin{aligned}
\frac{\partial\text{loss}(w,b)}{\partial w} &= \frac{\partial \frac{1}{n}\sum_i (wx_i + b - t_i)^2}{\partial w} \\
&= \frac{1}{n}\sum_i \frac{\partial (wx_i + b - t_i)^2}{\partial w} \\
&= \frac{1}{n}\sum_i \frac{\partial (wx_i + b - t_i)^2}{\partial (wx_i + b - t_i)} \frac{\partial (wx_i + b - t_i)}{\partial w} \\
&= \frac{2}{n}\sum_i (wx_i + b - t_i)x_i \\
\frac{\partial\text{loss}(w,b)}{\partial b} &= \frac{\partial \frac{1}{n}\sum_i (wx_i + b - t_i)^2}{\partial b} \\
&= \frac{2}{n}\sum_i (wx_i + b - t_i)
\end{aligned}$$

Here are the derivations of the two partial derivatives:

- first we use the *sum rule*, moving the derivative inside the sum symbol
- then we use the *chain rule*, to split the function into the composition of computing the residual and squaring, computing the derivative of each with respect to its argument.

The second homework exercise, and the formula sheet both provide a list of the most common rules for derivatives.

On your first pass through the slides, it's ok to take my word for it that these are the derivatives and to skip the derivation. However, there are a lot more derivations like these coming up, so you should work through every step before moving on to the next lecture, or you'll struggle in the later parts of the course.



gradient descent for our example

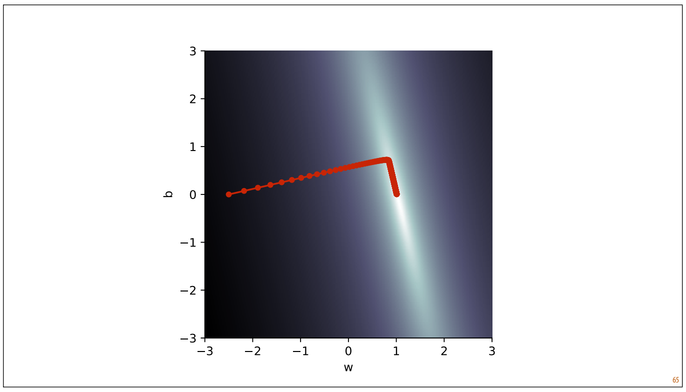
pick a random point (w, b) in the model space

loop:

$$\underbrace{\begin{pmatrix} w \\ b \end{pmatrix}}_{\text{next guess}} \leftarrow \underbrace{\begin{pmatrix} w \\ b \end{pmatrix}}_{\text{current guess}} - \eta \underbrace{\begin{pmatrix} \frac{2}{n}\sum_i (wx_i + b - t_i)x_i \\ \frac{2}{n}\sum_i (wx_i + b - t_i) \end{pmatrix}}_{\text{gradient}}$$

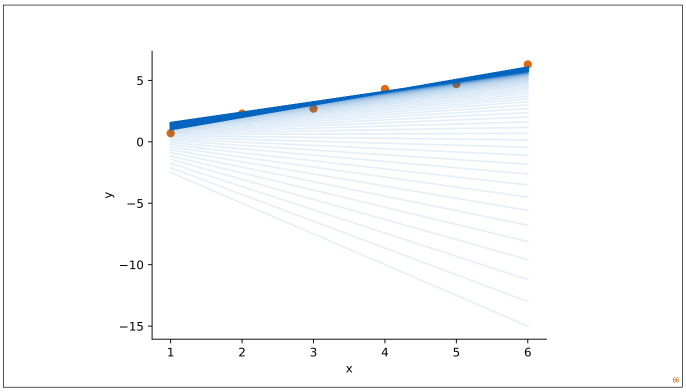
Here's what we've just worked out. Gradient descent, but specific to this particular model. We start with some initial guess, compute the gradient of the loss with the two functions we've just worked out, and we subtract that vector (times some scalar η) from our current guess.

Hopefully, repeating this process a number of times in small steps will directly follow the loss surface down to a (local) minimum.

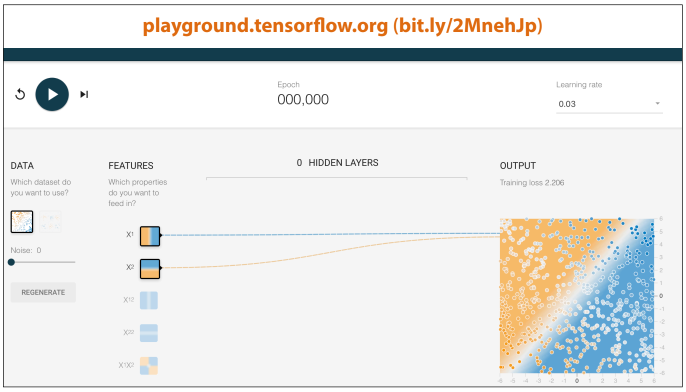


Here is the result on our dataset. Note how the iteration converges directly to the minimum. Note also that we have no *rejections* anymore. The algorithm is fully deterministic: it computes the optimal step, and takes it. There is no trial and error.

Note also that the gradient gives us a direction and a **step size**. As we get closer to the minimum, the function *flattens out* and the magnitude of the gradient decreases. The effect is that as we approach the minimum, the algorithm automatically takes smaller and smaller steps, preventing us from overshooting the optimum.



Here is what it looks like in feature space.



playground.tensorflow.org (bit.ly/2MnehJp)

Epoch
000,000

Learning rate
0.03

DATA
Which dataset do you want to use?

Noise: 0

FEATURES
Which properties do you want to feed in?

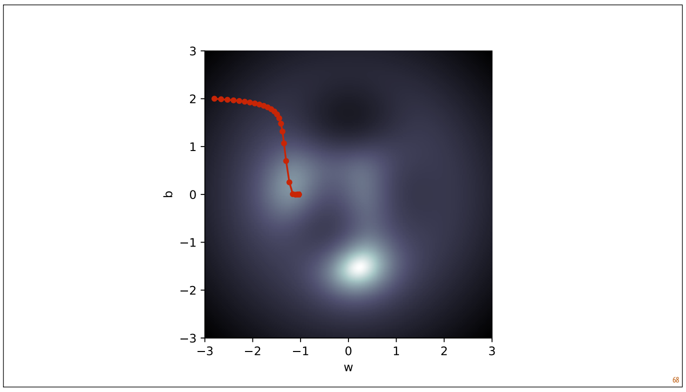
0 HIDDEN LAYERS

OUTPUT
Training loss 2.206

Here is a very helpful little browser app that we’ll return to a few times during the course. It contains a few things that that we haven't discussed yet, but if you remove all hidden layers, and set the target to regression, you'll get a linear classifier of the kind that we've been discussing. Click the following link to see a version with only the currently relevant features: **playground.tensorflow.com** We will enable different additional features as we discuss them in the course.

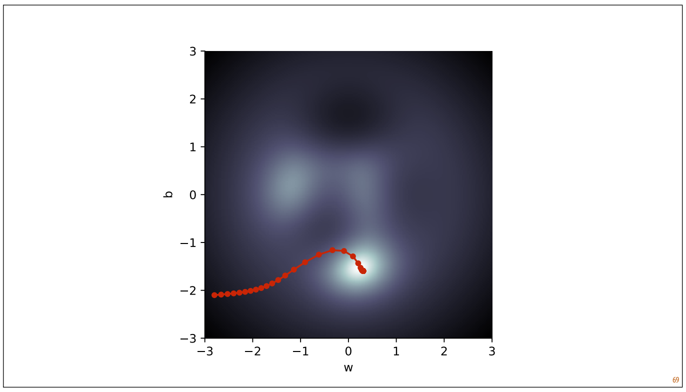
The output for the data is indicated by the color of the points, the output of the model is indicated by the colouring of the plane.

Note that the page calls this model a neural network (which we won’t discuss for a few more weeks). Linear models are just a very simple neural network.



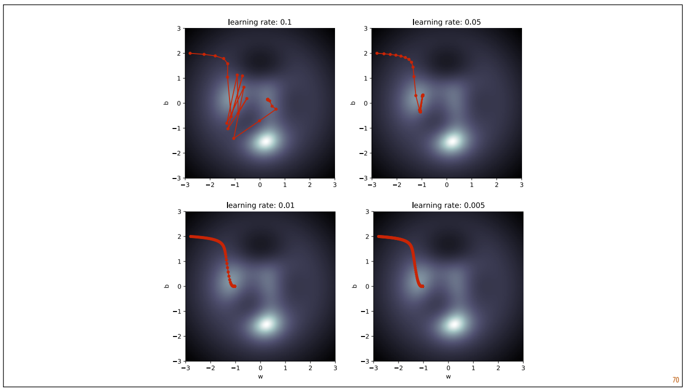
If our function is non-convex, gradient descent doesn't help us with the problem of local minima. As we see here, it heads straight for the nearest minimum and stays there. To make the algorithm more robust against this type of thing, we need to add a little randomness back in, preferably without destroying the behaviour of moving so cleanly to a minimum once one is found.

We can also try multiple runs from different starts. Later we will see *stochastic* gradient descent, which computes the gradient only over subsets of the data (making the algorithm more efficient, and adding a little randomness at the same time).



Here is a run with a more fortunate starting point.

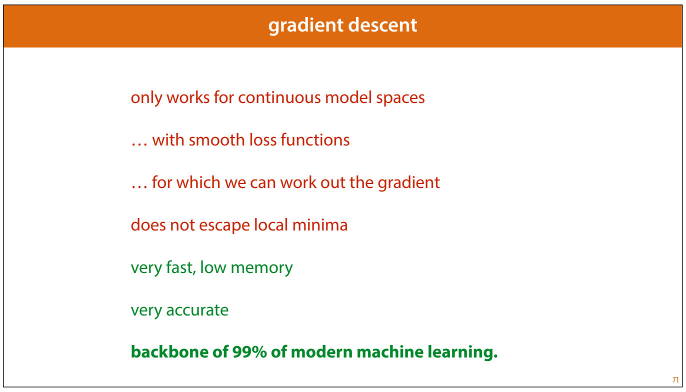
The point of convergence seems a little off in these images. The partial derivatives for this function are very complex (I used ***Wolfram Alpha*** *to find them), so most likely, the implementation has some numerical instability.*



Here, we see the effect of the learning rate. If we set if too high, the gradient descent jumps out of the first minimum it finds. A little lower and it stays in the neighborhood of the first minimum, but it sort of bounces from side to side, only very slowly moving towards the actual minimum.

At 0.01, we find a sweet spot where it finds the local minimum pretty quickly. At 0.005 we see the same behavior, but we need to wait much longer, because the step sizes are so small.

The best value of the learning rate is different for each dataset and each model. You'll usually have to find it by trial and error. We'll talk a little more about how this looks in practice in the next lecture.



gradient descent

only works for continuous model spaces

… with smooth loss functions

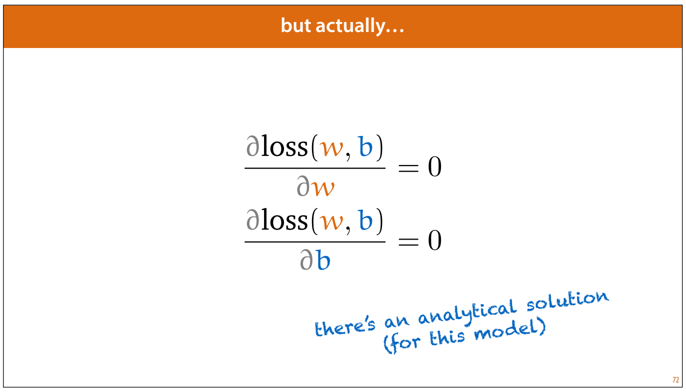
… for which we can work out the gradient

does not escape local minima

very fast, low memory

very accurate

backbone of 99% of modern machine learning.



but actually…

$$\frac{\partial \text{loss}(w, b)}{\partial w} = 0$$

$$\frac{\partial \text{loss}(w, b)}{\partial b} = 0$$

there's an analytical solution
(for this model)

It's worth saying that for **linear regression**, although it makes a nice, simple illustration, none of this *searching* is actually necessary. For linear regression, we can set the derivatives equal to zero and solve explicitly for **w** and for b. This would give us the optimal solution directly without searching.

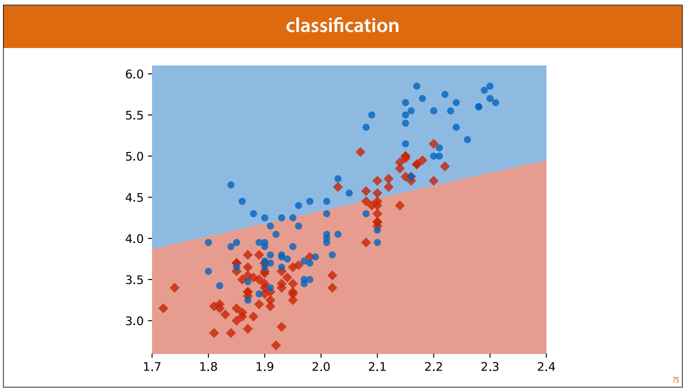
However, this trick requires more advanced linear algebra to work out than we want to introduce here. You should learn about this in most linear algebra courses, where the problem is called ordinary least squares*, and is solved by computing the pseudo-inverse of the data matrix. We won't go down this route in this course because it'll stop working very quickly once we start looking at more complicated models.*

Linear Models and Search

Part 4: Gradient descent and classification

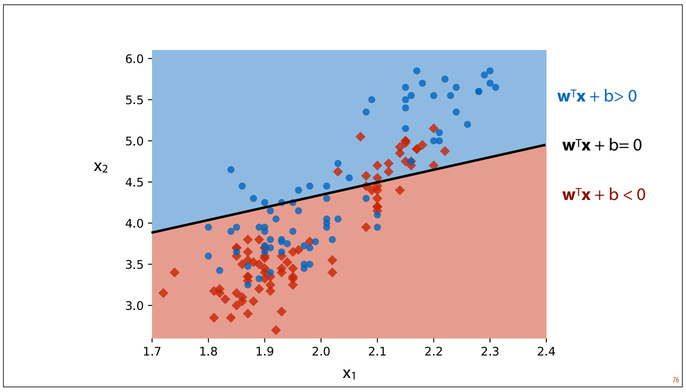
Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

|section|Gradient Descent and Classification|
|video|https://surfdrive.surf.nl/files/index.php/s/xpcqJFUDWMy4E3N/download|



Now, let's look at how this works for classification.

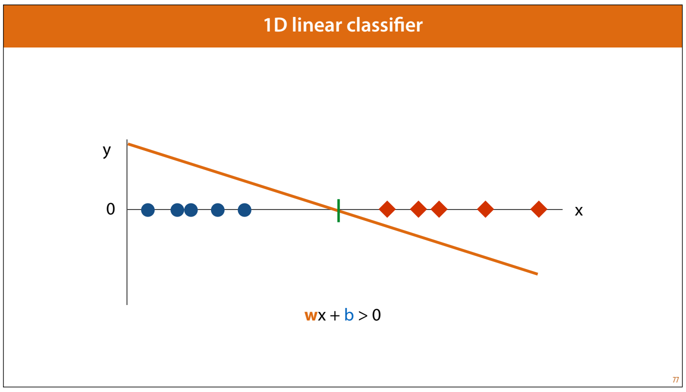
The first question we need to answer is how do we *define* a linear classifier: that is, a classifier whose decision boundary is always a line (or hyperplane) in feature space.



To define a linear decision boundary, we take the same functional form we used for the linear regression: some weight vector **w**, and a bias b.

The way we define the decision boundary is a little different than the way we defined the regression line. Here, we say that if $\mathbf{w}^T\mathbf{x}$ + b is larger than 0, we call **x** one class, if it is smaller than 0, we call it the other (we'll stick to binary classification for now).

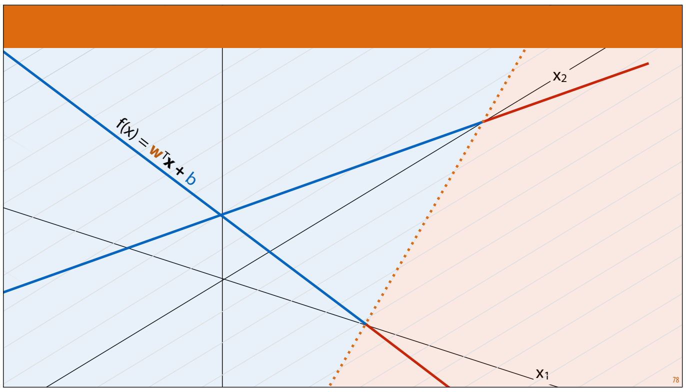
Note that we are drawing a line again, but in a different space: in the regression example we draw a line in the combined feature and output space (a function from the feature to the output). Here, we have two features, and we are drawing a line in only the feature space.



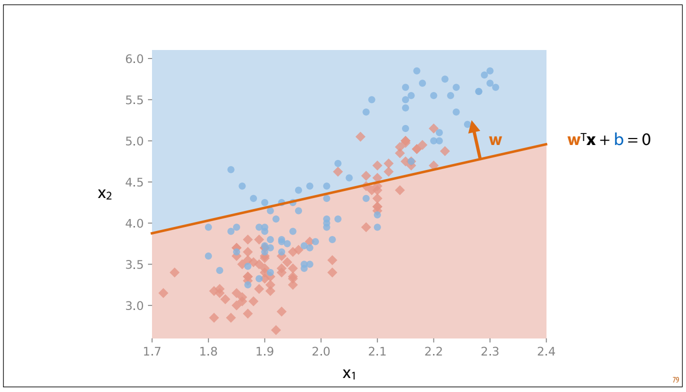
1D linear classifier

The actual hyperplane this function y = $\mathbf{w}^T\mathbf{x} + b$ defines can be thought of as lying above and below the feature space.

Here it is visualized for the case of one feature. We are defining a linear function from the feature to some output y. Wherever this line lies above the feature space (i.e. is positive), we classify things as the blue/disc class, and wherever the line lies below the feature space (i.e. is negative) we classify them as the red/diamond class.

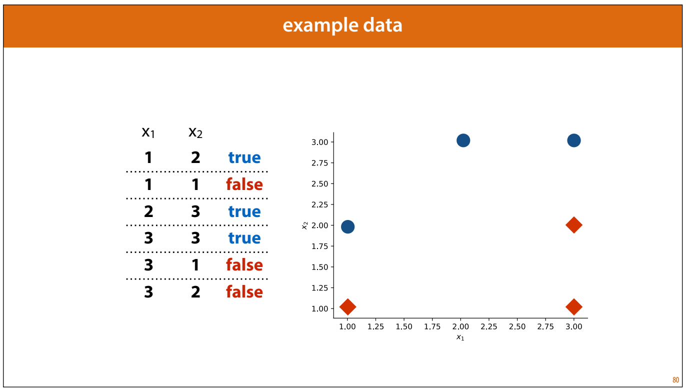


Here it is in 2D: $\mathbf{w}^T\mathbf{x} + b$ describes a plane that **intersects** the feature space. The line of intersection is our decision boundary.



This also shows us another interpretation of **w**. Since it is the direction of steepest ascent on this hyperplane, it is the vector **perpendicular to the decision boundary**, pointing to the class we assigned to the case where $\mathbf{w}^T\mathbf{x} + b$ is larger than 0 (the blue class in this case).

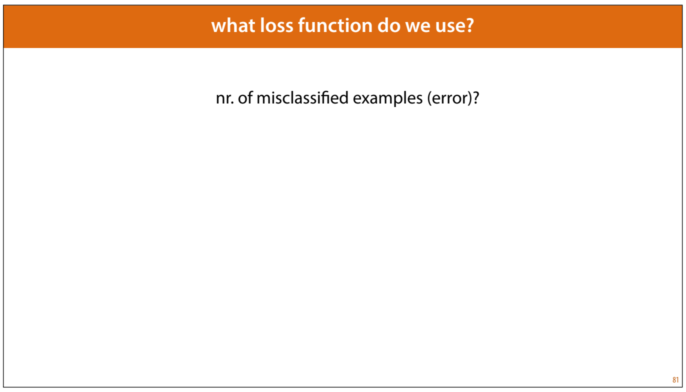
*We never want to "ascend" this plane like we do with the hyperplane approximating the loss landscape, but it's useful for our geometric intuition to know where **w** points, relative to our decision boundary. We will use this fact at different points in the future.*



example data

x_1	x_2	
1	2	true
1	1	false
2	3	true
3	3	true
3	1	false
3	2	false

Here is a simple classification dataset, which we'll use to illustrate the principle.

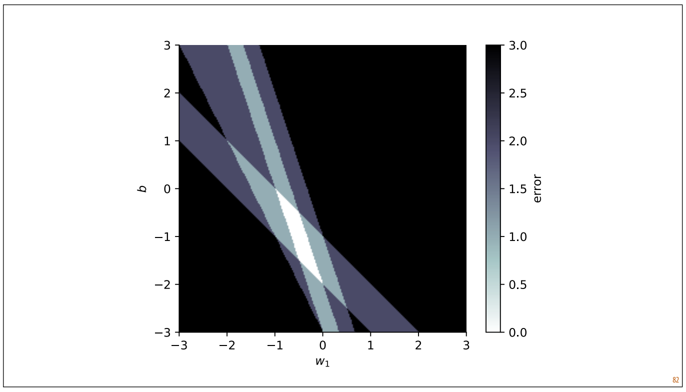


what loss function do we use?

nr. of misclassified examples (error)?

This gives us a model space, but how do we decide the quality of any particular model? What is our **loss function** for classification?

The thing we are usually trying to minimize is the **error**: the number of misclassified examples. Sometimes we are looking for something else, but in the simplest classification problems, this is what we are ultimately interested in: a classifier that makes as few mistakes as possible. So let's start there: *can we use the error as a loss function?*

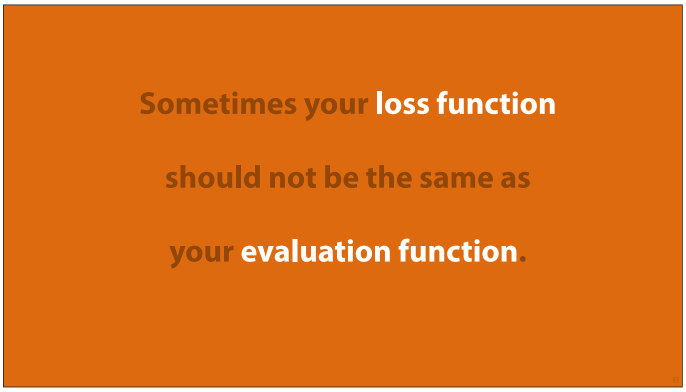


This is what our loss surface looks like for the error function on our simple dataset. Note that it consists almost entirely of flat regions. This is because changing a model a tiny bit will usually not change the number of misclassified examples. And if it does, the loss function will suddenly jump a lot.

In these flat regions, random search would have to do a random walk, stumbling around until it finds a ridge by accident.

Gradient descent would fare even worse: the gradient is zero everywhere in this picture, except exactly on the ridges, where it is undefined. Gradient descent would either crash, or simply never move.

Note that our model now has three parameters w_1, w_2 and b, so the loss surface is a function on a 3d space (a 4d "surface"). In order to plot it in two dimensions, we have fixed w_2=1.



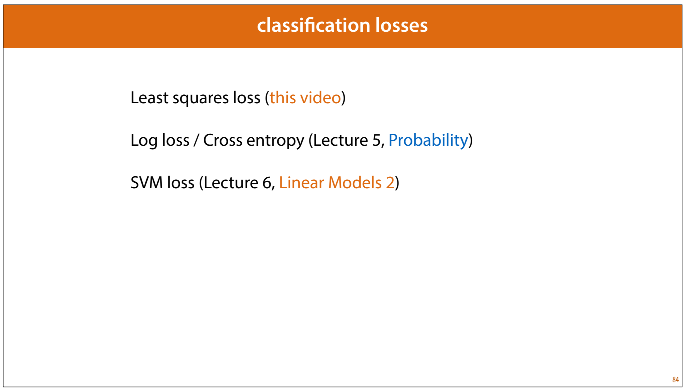
Sometimes your **loss function** should not be the same as your **evaluation function**.

This is an important lesson about loss functions. They serve two purposes:

1. To express what quantity we want to maximize in our search for a good model.
2. To provide **a smooth loss surface**, so that we can find a path from a bad model to a good one.

For this reason, it’s common not to use the error as a loss function, even when it’s the thing we’re actually interested in minimizing. Instead, we’ll replace it by a loss function that has its minimum at (roughly) the same model, but that provides a smooth, differentiable loss surface.

After we have trained a model we can still *evaluate* it with the function we're actually interested in (that is, we can still count how many mistakes it makes). We'll discuss evaluation in-depth in the next lecture.



classification losses

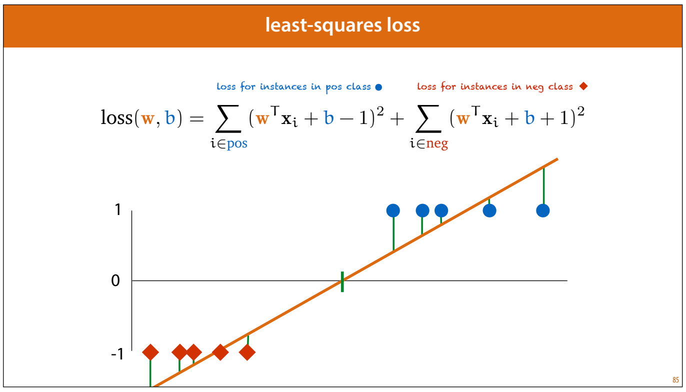
Least squares loss (this video)

Log loss / Cross entropy (Lecture 5, Probability)

SVM loss (Lecture 6, Linear Models 2)

In this course, we will investigate three common loss functions for classification. The first, least-squares loss, is just an application of MSE loss to classification, we will discuss that in the remainder of the lecture. It's not usually that good, but it gives you an idea of what a classification loss might look like.

The others require a bit more background, so we’ll save them for later.



least-squares loss

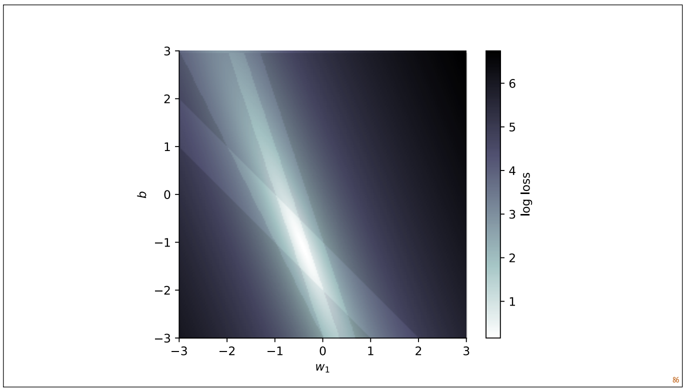
loss for instances in pos class ● loss for instances in neg class ◆

$$\text{loss}(\mathbf{w}, b) = \sum_{i \in \text{pos}} (\mathbf{w}^T \mathbf{x}_i + b - 1)^2 + \sum_{i \in \text{neg}} (\mathbf{w}^T \mathbf{x}_i + b + 1)^2$$

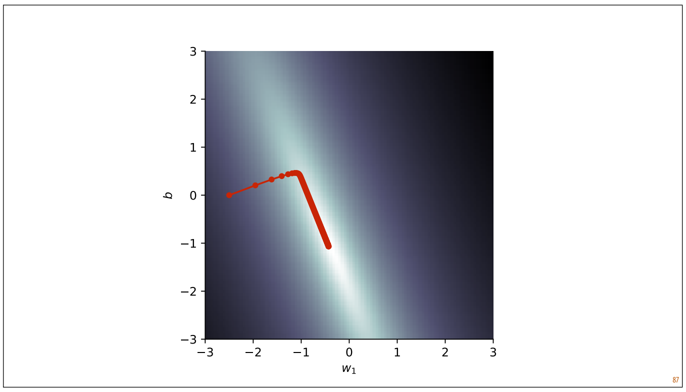
The least squares classifier essentially turns the classification problem into a regression problem: it assigns points in one class the numeric value +1 and points in the other class the value -1. We then use a basic MSE loss that we saw before the break to train a regression model to predict these numeric values.

Performing gradient descent with this loss function will result in a line that minimizes the green residuals. Hopefully the points are far enough apart that the decision boundary (the **single point** where the orange line crosses the x axis) separates the two classes.

As you can see, we always get very big residuals whatever we do. That is because the points simply do not lie on a single line, so the linear model is not appropriate. Still, with a little luck, the best fitting line will be positive for the +1 class and negative for the -1 class. If so the classifier will make the right predictions, even if the model is way off as a regression model for the numeric labels we introduced.

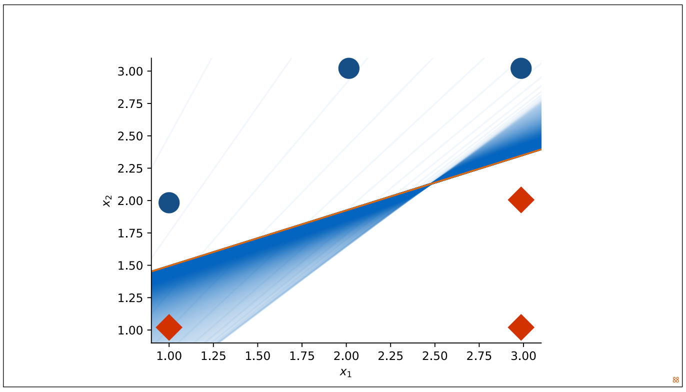


With this loss function, we note that our loss surface is perfectly smooth. If we overlay the error loss, we see that the minima of the two losses coincide pretty well (for this dataset at least).



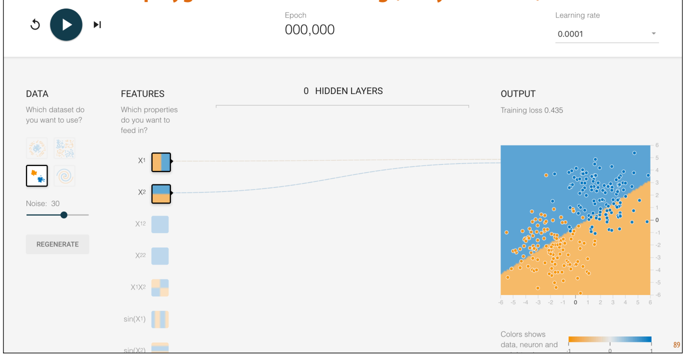
And gradient descent has no problem finding a solution.

Note, however that the optimum under this loss function may not always perfectly separate the classes, even if they are linearly separable. It does in our case, but this result is not guaranteed.



Here is the result in feature space, with the final decision boundary in orange.

playground.tensorflow.org (bit.ly/2Me1fxU)



Epoch
000,000

Learning rate
0.0001

DATA
Which dataset do you want to use?

Noise: 30

REGENERATE

FEATURES
Which properties do you want to feed in?

0 HIDDEN LAYERS

OUTPUT
Training loss 0.435

Colors shows data, neuron and

The tensorflow playground also allows us to play around with linear classifiers. Note that only for one of the two datasets, the linear decision boundary is appropriate.

Here is **a link with the relevant features enabled**.

This example actually uses a logarithmic loss, rather than a least squares loss, but it should still be intructive to play around with it. We'll discuss the logarithmic loss in the first probability lecture.

mlcourse@peterbloem.nl